AFRL-IF-RS-TR-2002-85
Final Technical Report
April 2002

FORMALLY VERIFIED HARDWARE ENCAPSULATION MECHANISM FOR SECURITY, INTEGRITY, AND SAFETY

SRI International

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D855

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

20020703 018

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-85 has been reviewed and is approved for publication.

APPROVED: GLEN E. BAHR
Project Engineer

FOR THE DIRECTOR: WARREN H. DEBANY, Jr., Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Apr 02	Final Aug 96 - Nov 01

4. TITLE AND SUBTITLE
FORMALLY VERIFIED HARDWARE ENCAPSULATION MECHANISM FOR SECURITY, INTEGRITY, AND SAFETY

5. FUNDING NUMBERS
C - F30602-96-C-0204
PE - 62301E
PR - D855
TA - 00
WU - 01

6. AUTHOR(S)
John Rushby

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025-3493

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

AFRL/IFGB
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-85

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Glen E. Bahr, IFGB, 315-330-3515

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Safety- and security-critical systems both require encapsulation of code and data belonging to different applications and sensitivity levels. It must be impossible for a fault or Trojan Horse in one application to affect the operation or real-time performance of another, or for information of one sensitivity level to contaminate that of another. Encapsulation is achieved by the combination of software in an operating system kernel managing hardware protection mechanisms. The critical nature of the applications concerned means that extremely high assurance is required for the correctness of the encapsulation mechanisms. A systematic approach is developed for the formal specification and verification of these encapsulation properties and mechanisms, addressing the interaction between a processor, custom protection hardware, and the kernel software managing them; it encompasses both safety and security concerns and may be adjusted for different classes of systems. It is validated by mechanically checked verification. This validation provides a formal guarantee--from kernel interface down through hardware--of both spatial (memory) and temporal (time-slicing) encapsulation for the processor.

14. SUBJECT TERMS
computer code encapsulation, hardware verification, custom protection hardware, kernel software, safety, security, AAMP-FV processor, Advanced Architecture Microprocessor, invariant performance, Schultz model

15. NUMBER OF PAGES
296

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Contents

I Introduction 1
Project Summary and Organization of the Report 2

II Requirements, Mechanisms, Assurance, and Architectures for Encapsulation 7
Partitioning for Safety and Security 9
Partitioning System Requirements and Architecture 78

III Strict Encapsulation: A Precise Characterization and An Application 95
Invariant Performance 97
Invariant Performance and PVS 107

IV Methodologies and Tools to Automate Verification of Encapsulation 124
Verifying Advanced Microarchitectures
that Support Speculation and Exceptions 126
Automating Partition Proofs 143
Verification Diagrams Revisited:
Disjunctive Invariants for Easy Verification 158
Static Analysis for Safe Destructive Updates 174
PVS Code Proofs: Benchmarking and Enhancements 190
Ubiquitous Abstraction:
A New Approach for Mechanized Formal Verification 199
An Overview of SAL 203
Symbolic Analysis of Transition Systems 214
Combining Theorem Proving and Model Checking
through Symbolic Analysis 233

V Technology Transition 252
Invariant Performance and PMU Design Considerations 254
Invariant Performance and Commercial System Components 263

Bibliography 287

Part I

Introduction

Project Summary and Organization of the Report

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

1 Objective

Safety- and security-critical systems such as integrated avionics and encryption controllers both require encapsulation of code and data belonging to different applications and sensitivity levels: even though a system is shared by several applications, it must be impossible for a fault or Trojan Horse in one application to affect the operation or real-time performance of another, and it must be impossible for information of one sensitivity level to contaminate that of another. Encapsulation is achieved by the combination of software in an operating system kernel managing hardware protection mechanisms (e.g., memory management and watchdog timers). The critical nature of the applications concerned means that extremely high assurance is required for the correctness of the encapsulation mechanisms. This project develops and demonstrates the use of mechanically checked formal verification methods to provide this assurance.

2 Approach

The security and safety communities have both developed requirements that specify how applications may share resources without disrupting one another, and that prescribe how the applications may communicate exclusively through known channels. The two fields have evolved different approaches that may be synthesized into a common set of requirements, issues, and implementation mechanisms, yielding a cross-fertilization that should reduce costs and increase assurance for future acquisitions.

A systematic approach is developed for the formal specification and verification of these encapsulation properties and mechanisms. The approach addresses the interaction between a processor, custom protection hardware, and the kernel software that manages these; it encompasses both safety and security concerns and may be adjusted for the characteristics of different classes of systems.

The approach is validated by mechanically checked verification of a design derived from the Collins AAMP-FV processor, whose microarchitecture and microcode have been formally verified in a previous project [3]. This validation provides a formal guarantee—from kernel interface down through hardware—of both spatial (memory) and temporal (time-slicing) encapsulation for a processor of the kind used for life-critical avionics functions.

3 Accomplishments

Work accomplished in the first two years of this contract defined the space of design choices for encapsulation in safety-critical architectures, including consideration of

partitioning mechanisms for security and other critical properties; identification of interactions between these mechanisms and those for system structuring, scheduling, and fault tolerance; and discussion of issues involved in providing assurance for partitioning. The initial work also proposed a set of noninterference properties referred to as *invariant performance*, and formalized a corresponding conjecture for the *Schultz model*, a simple implementation of a small kernel and supporting hardware that maintains noninterfering partitions on an enhanced Collins AAMP-FV. The final two years of contract work completed and extended these results, including mechanized proofs of noninterference properties of the Schultz model; application of invariant performance to the design and formalization of a real-time, safety-critical, partitioned system; evaluation of the utility of invariant performance for partitioning of commercial components in certified systems; and the development and documentation of methodologies and tools to mechanize the verification of encapsulation properties. A summary of the accomplishments produced in the course of this contract appears below, along with a guide to the accompanying documents that constitute the body of this final report.

1. We have documented a comprehensive analysis of issues in the specification, design, and implementation of encapsulation mechanisms for safety-critical applications. Appended report, Part II: *Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance* by John Rushby, SRI International.

2. We have refined a set of requirements and specified a correlative architecture for partitioning in avionics systems. Appended report, Part II: *Partitioning System Requirements and Architecture* by David Hardin, Rockwell Collins.

3. We have developed the concept of *invariant performance* that provides a contract with application developers and a basis for independent verification and validation of the safety and security of applications run on a shared platform. Appended report, Part III: *Invariant Performance* by Matthew Wilding, Rockwell Collins.

4. We have completed a user partition memory protection proof for Schultz using deductive methods in PVS. The proof establishes both spatial and temporal encapsulation and involves 900 subproofs, including subsidiary type correctness conditions and sublemmas. Appended report, Part III: *Invariant Performance and PVS: A Statement of Task Isolation that can be Certified by Machine-Checking* by David Greve and Matthew Wilding, Rockwell Collins.

5. We have developed and documented several novel methodologies for the verification of encapsulation mechanisms that satisfy invariant performance and related properties. We have also developed and documented general methods, such as "Disjunctive Invariants" and the "Completion Functions Ap-

proach," that have the potential to improve the mechanization and scalability of such proofs. Appended reports and papers, Part IV: *Verifying Advanced Microarchitectures that Support Speculation and Exceptions* by Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas, SRI International; *Automating Partition Proofs* by Mandayam Srivas, SRI International; *Verification Diagrams Revisited: Disjunctive Invariants* by John Rushby, SRI International.

6. We have implemented enhancements to PVS to support efficient automation of the proofs required to establish properties of encapsulation architectures, including static analysis to determine safe destructive array updates. Appended report, Part IV: *Static Analysis for Safe Destructive Updates* by N. Shankar, SRI International.

7. We have developed interfaces to allow SAL (Symbolic Analysis Laboratory) to interoperate with other tools and languages, including PVS, SMV, JAVA, and InVeSt [1]. In the context of SAL, we have augmented PVS with tools for abstraction, invariant generation, and program analysis. We have also developed techniques for effectively combining abstraction with deductive and algorithmic proof methods within the SAL framework. Appended reports, Part IV: *An Overview of SAL* by Saddek Bensalem et al., SRI International; *Ubiquitous Abstraction: A New Approach for Mechanized Formal Verification*, by John Rushby, SRI International; *Symbolic Analysis of Transition Systems* by N. Shankar, SRI International; *Combining Theorem Proving and Model Checking through Symbolic Analysis* by N. Shankar.

8. We have extended the concept of invariant performance to the JEM2/PMU, a real-time, safety-critical system that exploits a virtual machine partitioning scheme to support high-rate context switching. We have constructed executable formal models of the JEM2 and the JEM2 PMU that allow us to generate functional simulators to validate these design models. JEM2 model validation has also used execution of test programs, and the PMU model has been integrated with a VHDL simulation environment to provide a PMU test bench circuit. Appended report, Part V: *Invariant Performance and PMU Design Considerations* by David Greve, Rockwell Collins.

9. We have demonstrated the utility of the notion of invariant performance for system certification, and developed techniques of automated modeling and reasoning that have been integrated into the Rockwell Collins computer system development cycle [2]. Appended report, Part V: *Invariant Performance and Commercial System Components* by David Greve, Rockwell Collins.

4 Organization of the Report

The organization of this final report reflects the main topics identified in the preceding summary. In order of presentation, the major sections are as follows: issues in the specification, design, and implementation of encapsulation mechanisms and an architecture that addresses them; a precise characterization of strict encapsulation; methodology and tools; and technology transition.

References

[1] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, pages 505–510, Vancouver, Canada, June 1998.

[2] David Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, Volume 1522 of Springer-Verlag *Lecture Notes in Computer Science*, pages 321–333, Palo Alto, CA, November 1998.

[3] Steven Miller, David Greve, Matthew Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Cedar Rapids, IA, 1996.

Part II

Requirements, Mechanisms, Assurance, and Architectures for Encapsulation

Overview

Automated aircraft control traditionally implemented distinct functions as separate systems (e.g., autopilot, autothrottle, flight management) that inherently contained faults within the system in which they occurred. By contrast, modern avionics architectures typically support multiple functions on a single, shared, fault-tolerant computer system in which fault containment is intrinsically less well defined. The approach to strong fault containment proposed in our work uses hardware and software mechanisms to enforce a *partitioning* regime. These mechanisms are analogous to those used in computer security to assure *separation* of information of differing sensitivity levels.

Our DARPA contract work identifies requirements for partitioning in safety-critical applications, explores mechanisms for achieving those requirements with very high assurance, defines an architecture that exploits these mechanisms to satisfy the given requirements, and studies the parallels between these partitioning mechanisms and separation mechanisms used in the computer security domain [8, 11].

Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

Automated aircraft control has traditionally been divided into distinct "functions" that are implemented separately (e.g., autopilot, autothrottle, flight management); each function has its own fault-tolerant computer system, and dependencies among different functions are generally limited to the exchange of sensor and control data. A by-product of this "federated" architecture is that faults are strongly contained within the computer system of the function where they occur and cannot readily propagate to affect the operation of other functions.

More modern avionics architectures contemplate supporting multiple functions on a single, shared, fault-tolerant computer system where natural fault containment boundaries are less sharply defined. *Partitioning* uses appropriate hardware and software mechanisms to restore strong fault containment to such integrated architectures.

In computer security, information of different sensitivity levels must be kept separate and the requirements for highly assured *separation* are similar to those for partitioning. It is possible that the mechanisms and techniques developed for computer security can contribute to those required for avionics in particular, and safety in general, and vice-versa.

Since partitioning for safety is less well studied than separation for security, this report examines the requirements for partitioning in avionics, mechanisms for their realization, and issues in providing assurance for them. Security models are then reviewed and compared with the concerns of partitioning.

Chapter 1

Motivation and Introduction

Digital flight-control is a quintessential safety-critical application. Digital flight-control functions in current aircraft are generally implemented by a *federated* architecture in which each function (e.g., autopilot, flight management, yaw damping, displays) has its own computer system that is only loosely coupled to the computer systems of other functions. A great advantage of this architecture is that fault containment is inherent: that is to say, a fault in the computer system supporting one function, or in the software implementing that function, is unlikely to propagate to other functions because there is very little that is shared across the different functions. To be sure, some functions interact with others, but these interactions are accomplished by the exchange of data, and functions can be designed to detect and tolerate a faulty or erratic data source.

The obvious disadvantage to the federated approach is its profligate use of resources: each function needs its own computer system (which is generally replicated for fault tolerance), with all the attendant costs of acquisition, space, power, weight, cooling, installation, and maintenance. Integrated Modular Avionics (IMA) has therefore emerged as a design concept to challenge the federated architecture [1,78]. In IMA, a single computer system (with internal replication to provide fault tolerance) provides a common computing resource to several functions. As a shared resource, IMA has the potential to diminish fault containment between functions: for example, a faulty function might monopolize the computer or communications system, denying service to all the other functions sharing that system, or it might corrupt the memory of other functions, or send inappropriate commands to their actuators. It is almost impossible for individual functions to protect themselves against this kind of corruption to the computational resource on which they depend, so any realization of IMA must provide *partitioning* to ensure that the shared computer system provides protection against fault propagation from one function to another that is equivalent to that which is inherent to the federated architecture.

The purpose of this report is to identify the requirements for partitioning in safety-critical applications, to explore topics in achieving those requirements with very high assurance, and to relate these to similar issues in computer security.

We use IMA as our canonical safety-critical application. The next chapter, therefore, is concerned with the general requirements for IMA and partitioning, and the one following with issues in the implementation of IMA and the mechanisms for partitioning. In Chapter 4, we consider methods developed for specifying and analyzing computer security policies and relate these to the needs of partitioning in avionics. We end with conclusions and suggestions for future work.

Chapter 2

Informal Requirements

To gain insight into the requirements for partitioning, we first need to examine the context provided by IMA and related developments in avionics.

2.1 Integrated Modular Avionics

It can be argued that the simplest interpretation of IMA envisions an architecture that technology has already rendered obsolete: an embedded systems version of the centralized time-shared "mainframe." Thanks to recent technological developments, powerful processors, large memories, and high-bandwidth local communications are all available as reliable and inexpensive commodity items, and these developments surely favor *less* rather than more centralization. Thus, this argument proceeds, a modern avionics architecture should be more, not less, federated, with existing functions "deconstructed" into smaller components, each having its own processor.

There is some plausibility to this argument, but the distinction between the "more federated" architecture and centralized IMA proves to be moot on closer inspection. A federated architecture is one whose components are very loosely coupled—meaning that they can operate largely independently. But the different elements of a function—for example, vertical and horizontal flight path control in an autopilot—usually are rather tightly coupled (and it is argued below that they should become even more tightly coupled), so that the deconstructed function would not be a federated system so much as a *distributed* one—meaning a system whose components may be physically separated, but which must coordinate to achieve some collective purpose. Dually, a centralized IMA architecture would not be a simple mainframe—for a computer system supporting flight functions must provide replicated and physically distributed hardware for fault tolerance, together with mechanisms for redundancy management. Consequently, a conceptually centralized architecture will be, internally, a distributed system, and the basic services that

it provides will not differ in a significant way from those required for the "more federated" architecture.

Another contrarian point of view is that neither centralized IMA nor the "more federated" architecture offers significant benefits over current practice; the present federated architecture has been validated by experience, and modern hardware technology will reduce its cost penalty—so there is no reason to change it. The argument against this point of view is that it takes a very narrow interpretation of the costs associated with the current architecture, and therefore grossly underestimates them. One neglected cost is safety: the federated architecture has the advantage of natural fault containment, but it imposes a cost in poorly coordinated control, and complex and fault-prone pilot interfaces.

The current allocation of flight automation to separate functions is the result of largely accidental historical factors. Consequently, certain control variables that are tightly coupled in a dynamical sense are managed by different functions: for example, engine thrust is managed by the autothrottle, and pitch angle by the autopilot. Since a change in either of these variables affects the other, but there is no higher-level function that manages them in a coordinated manner, such conceptually simple services as "cruise speed control," "altitude select," and "vertical speed" have complex and imperfect implementations that are difficult to manage. For example, Lambregts [59, page 4] reports:

> "Because the actions of the autothrottle are not tactically coordinated with the autopilot, the autothrottle speed control constantly upsets the autopilot flight path control and vice versa, resulting in a notorious coupling problem familiar to every pilot. It manifests itself especially when excited by turbulence or windshear, to the point where the tracking performance and ride quality becomes unacceptable. The old remedy to break the coupling was to change the autopilot mode to ALTITUDE HOLD (e.g., the older B747-200/300). On newer airplanes, this problem has been reduced to an acceptable level for the cruise operation after a very difficult and costly development process, implementing provisions such as separation of the control frequency by going to very low autothrottle feedback gain, application of 'energy compensation,' turbulence compensation, and nonlinear windshear detections/compensation."

And again:

> "Due to the lack of proper control coordination, the autopilot ALTITUDE SELECT and VERTICAL SPEED modes never functioned satisfactorily... these problems resulted in development of the FLIGHT LEVEL CHANGE (FLC) mode that was first implemented on the B757/B767... however the mode logic depends on certain assumptions

> that are valid only for certain operations, so the logic can be tricked and cause an incorrect or poorly coordinated control response... as a result there have been a number of incidents where the FLC mode did not properly execute the pilot's command."

The lack of properly integrated control caused by the artificial separation of functions in the federated architecture is one of the factors that leads to the complex modes and submodes used in these functions, and thence to the "automation surprises" and "mode confusions" that characterize problems in the "flightcrew-automation interface." Numerous fatal crashes and other incidents are attributed to such human factors problems [28, Appendix D], but it is clear from their origins in the artificial separation of functions that these problems are unlikely to be solved by local improvements in the interfaces and cues presented to pilots. The plethora of modes, submodes, and their corresponding interactions also exacts a high cost in development, implementation, and certification. If this analysis is correct, the traditional federated architecture is a major obstacle to a more rational organization of flight functions, and IMA is the best hope for removing this obstacle.

The topics considered so far suggest that the appropriate context in which to examine partitioning for IMA is a distributed system in which flight functions (which might well be defined and subdivided differently than in the traditional federated architecture) are each allocated to separate processors (replicated as necessary for fault tolerance). In this model, we would need to consider partitioning to limit fault propagation *between* the processors supporting each function, but not *within* them. This model, however, overlooks a new opportunity that could be created by more fine-grained partitioning.

If functions have no internal partitioning, then all their software must be assured and certified to the level appropriate for that function. Thus, all the software in an autopilot function is likely to require assurance to Level A of DO-178B (this, the highest level of DO-178B, the guidelines for certification of airborne software [30,84], is for software whose malfunction could contribute to a catastrophic failure condition [29]), and this discourages the inclusion of any software that is not strictly essential to the function. While this may be a good thing in general, it also discourages inclusion of services that could have a positive safety impact, such as continuous self-test, or for-information-only messages to the pilot. More generally, partitioning within a processor could allow an individual function to be divided into software components of different criticalities; each could then be developed and certified to the level appropriate to its criticality, thereby reducing overall costs while allowing assurance effort to be focused on the most important areas. Without partitioning, the concern that a fault in less critical software could have an impact on the operation of more critical software necessarily elevates the criticality of the first to that of the second; partitioning would remove the danger of fault propagation, and allow the criticality of each software component to be assessed more locally.

The considerations of the previous paragraph suggest that for partitioning within a single processor it might be appropriate to limit attention to the case where the processor is shared by the components of only a single function. We might suppose that these components consist of one implementing the main function, and several others providing subsidiary services. Since a fault in the main component amounts to a fault in (this replica of) the overall function, there seems little point in protecting the subsidiary components from faults in the main component, and this suggests that partitioning could be asymmetric (the main component is protected from the subsidiary ones, but not vice versa). It is not clear whether such asymmetry would provide any benefit in terms of simplicity or cost of the partitioning mechanisms, but the point is probably moot since other scenarios require a symmetric approach. One scenario is support for several minor functions, for example undercarriage and weather radar, on a single processor. Where the functions are not required at the same time, partitioning could perhaps be achieved by giving each one sole command of its processor while it is active (this is similar to "periods processing" in the security context), but the more general requirement is for simultaneous operation with symmetric partitioning. The second scenario concerns very cost sensitive applications, such as single-engine general aviation aircraft. Here it may be desirable to run multiple major functions (such as autopilot and rudimentary flight management) on a single (possibly non-fault-tolerant) processor. There are even proposals to host these functions on mass-market systems such as Windows NT. Although one can be skeptical of this proposal (particularly if "free flight" air traffic control makes flight management data from general aviation aircraft critical to overall airspace safety), it seems worth examining the technical feasibility of symmetric partitioning for critical functions within a single processor.

The current federated architecture not only uses a lot of computer systems, it uses a lot of *different* computer systems: each function typically has its own unique computer platform. There is a high cost associated with developing and certifying software to run on these idiosyncratic platforms. Logically independent of IMA, but coupled to it quite strongly in practice, are moves to define standardized interfaces to the platforms that support flight functions, and to introduce some of the abstractions and services provided by an operating system. The ARINC 653 (APEX) [4] standard represents a step in this direction. Developments such as this could significantly reduce the cost of avionics software development and might stimulate creation of standard modules for common tasks that could be reused by different functions running on different platforms.

The design choices for partitioning interact with those for providing operating system services. The major decision is whether partitioning is provided above an operating system layer (Figure 2.1(a)), or above a minimal kernel (or executive) with most operating system services then provided separately in each partition (Figure 2.1(b)). The first choice is the way standard operating systems are structured

(with partitions being client processes), but it has the disadvantage that partitioning then relies on a great deal of operating system software. The second choice is sometimes called the "virtual machine" approach, and it has the advantage that partitioning relies only on the kernel and its supporting hardware.[1]

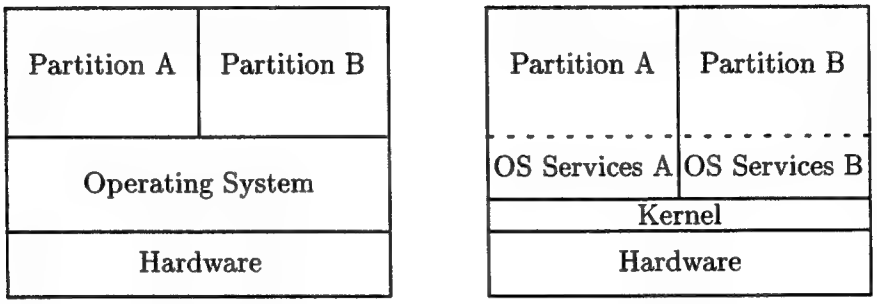


Figure 2.1: Alternative Operating System/Partitioning Designs

Another area where IMA has the potential to reduce costs is through improved dispatch reliability. Critical flight functions must tolerate hardware faults, and so they run on replicated hardware (typically quad-redundant or greater for primary flight control and displays, triple for autopilot and autoland, dual for flight management and yaw damping, and single for autothrottle). But despite the massive cost of providing a fault-tolerant platform for each function, and despite the large number of separate processors and other components available (there can be as many as 50 processors among the major functions of a large modern transport plane), the federated architecture does not provide a large margin of redundancy, nor operational flexibility. A single faulty processor in any function may be enough to prevent takeoff (thereby requiring maintenance in possibly less than ideal circumstances), and multiple faults afflicting such a function during flight might trigger a diversion, or have even more serious consequences. With IMA, in contrast, replicated processors are not bound to a specific function, but can be allocated as required: normal operation can continue as long as the total number of nonfaulty processors is sufficient to provide the required level of replication to each function. This increases

[1]Some operating systems use the second model. It was first employed in VM/370 [70], which served as the basis for a major early secure system development [7,36]. Fully virtualizing the underlying hardware is expensive, so later "μ-kernels" such as Mach and Chorus provided a more abstract interface. These also proved to have disappointing performance. Second-generation μ-kernels and comparable toolkits such as Exokernel [49], Flux [32], L4 [39], and SPIN [11] achieve good performance and introduce several implementation techniques relevant to the design of partitioned systems.

overall safety margins, while also allowing maintenance to be deferred (e.g., until the aircraft's schedule brings it to a major maintenance base) [41].[2]

The ability to exploit this increased redundancy and flexibility depends on a systematic approach to fault tolerance within functions (so that they are not tightly bound to a specific processor), and across the distributed coordination mechanisms of the IMA platform itself. Design of fault-tolerant systems is not only a massively difficult and expensive activity (the basic mechanisms of fault tolerance concern the coordination of distributed, real-time systems operating in the presence of faults, which are among the hardest problems in computer science) but is often a pervasive one: that is, mechanisms for fault tolerance and redundancy management in avionics are seldom encapsulated as an operating system or middleware service, and instead affect the design of every piece of software within the function. As a result, it is generally impossible to take software—or even the design for a piece of software—from one function and reuse it in another, or on another platform, even when standards such as APEX are used, for these standards concern only the mechanics of system calls and do not address the deeper concerns of systematic and transparent fault tolerance. Another reason for the pervasive influence of fault tolerance in current system designs is that the these mechanisms (and most others that involve coordination across multiple processors and functions) are seldom *compositional*—meaning that there is no *a priori* guarantee that elements that each work on their own will also work in combination. The massive resources expended on systems integration are a symptom of the lack of compositionality provided by current design practices.

Thus, full realization of the benefits of IMA requires adoption of modern concepts for systematic, compositional, fault-tolerant real-time system design [55]. These would reduce the pervasive impact of fault tolerance in avionics software development and provide cost savings and opportunities for reuse that could be much greater than those provided by lower-level standards such as APEX. Taken to their conclusion, such approaches could completely decouple the implementation of flight functions from that of their fault-tolerant platform, possibly enabling each to be certified separately. The impact of such developments on partitioning is, first, a requirement that the distributed partitioning mechanisms must themselves be robustly fault tolerant and, second, that these mechanisms must cooperate with operating system or kernel functions to provide the services required for systematic and transparent fault tolerance in the implementations of flight functions.

Summarizing this review of issues in IMA, we see that partitioning should be considered both within a single processor and across a distributed system, and that partitioning has interactions with the provision of operating system services and transparent fault tolerance. In the next section we examine the requirements for partitioning a little more closely.

[2]Current implementations of IMA allocate functions to processors at startup time; reconfiguration in flight is a future prospect.

2.2 Partitioning

The purpose of partitioning is fault containment: a failure in one partition must not propagate to cause failure in another partition. However, we need to be careful about what kinds of faults and failures are considered. The function in a partition depends on the correct operation of its processor and associated peripherals, and partitioning is not intended to protect against their failure—this can be achieved only by replicating functions across multiple processors in a fault-tolerant manner. After all, each function would be just as vulnerable to hardware failure if it had its own processor. Rather, the intent of partitioning is to control the additional hazard that is created when a function shares its processor (or, more generally, a resource) with other functions. The additional hazard is that faults in the design or implementation of one function may affect the operation of other functions that share resources with it.[3] Now a design or implementation fault in a flight function is surely a very serious event and it might be supposed that (a) such faults are so serious that it does not matter what else goes wrong, or (b) certification ensures that such faults cannot occur. Both suppositions would, if true, diminish the requirements for partitioning.

The first point is easily refuted: the whole thrust of aircraft certification is to ensure that failures are independent (and individually improbable) if their combination could be catastrophic. Thus, while a design fault in, say, the autothrottle function would be serious, appropriate design and system-level hazard analysis will ensure that it is not catastrophic, *provided* other functions do not fail at the same time. Allowing a fault in this function to propagate to another (e.g., autoland) would violate the assumption of independent failures. Thus, far from a fault in a critical function being so serious as to render concern for partitioning irrelevant, it is the need to contain the consequences of such a fault that renders partitioning essential (and elevates its criticality to at least that of the most critical function supported).

It could be argued that both functions will certainly be lost if their shared processor fails, so they surely would not be sharing if their correlated failure could be catastrophic. This overlooks a couple of points. First, malfunction or unintended function is often more serious than simple loss of function, and the consequences of a propagating fault (unlike those of a processor failure) may well be of these more serious kinds. For example, a buffer overflow in one function might overwrite data in another, leading to unpredictable consequences. (The Phobos I spacecraft was lost in just this circumstance—when a keyboard buffer overflowed into the memory of a critical flight control function [14, 17].) Second, the increased interdependency wrought by IMA may introduce shared resources—and hence paths for fault

[3] Partitioning can also limit the consequences of transient hardware faults (by containing them within the partition that is directly affected), but that is a side benefit, not a requirement.

propagation—that are less obvious and more easily overlooked than shared processors. For example, functions in separate processors, where correlated failure would not be anticipated (and would not occur in a federated architecture) might become vulnerable to fault propagation through a shared bus in an IMA architecture.

Returning to the second point raised above (that certification ought to ensure the absence of design and implementation faults), note that certification requires assurance proportional to the consequences of failure. In a federated architecture, such consequences are generally limited to the function concerned, so that assurance is related to the criticality of that function. But if the failure of one function could propagate to others, then a low-criticality (and correspondingly low assurance) function might cause a high-criticality function to fail. This means that either all functions that share resources must be assured to the level of the most critical (such elevation in assurance levels is directly contrary to one of the goals of IMA), or that partitioning must be used to eliminate fault propagation from low-assurance functions to those of high criticality. When different functions already happen to have the same level of assurance, the need for partitioning may not be so great, and it has been suggested that functions with software assured to Level A of DO-178B may be allowed to share resources without partitioning. Note, however, that a fault that causes one function to induce a failure in another might not affect the operation of the first (as noted above, a temporary buffer overflow can have this property). And although certification requires assurance of the absence of such unintended effects, as well as positive assurance that the intended function is performed correctly, it is generally much harder to provide the first kind of assurance than the second. Furthermore, shared resources create new pathways for the propagation of unintended effects, and these pathways might not have been considered when assurance was developed for the individual functions. Consequently, partitioning seems advisable even when the functions concerned are of the same level of criticality, and all software is assured to the same level.

Summarizing the discussion in this chapter, we may conclude that future avionics architectures will have the character of distributed, rather than federated, systems, and that multiple functions, of possibly different levels of criticality and assurance, will be supported by the same system. Resources, such as processors, communications buses, and peripheral devices, may be shared between different functions. Shared resources introduce new pathways for fault propagation, and these hazards must be controlled by partitioning.

Because partitioning is required to prevent fault propagation through shared resources, a suitable benchmark or "Gold Standard" for the effectiveness of partitioning would seem to be a comparable system (intuitively a federated one) in which there are no shared resources. This is captured in the following.

Gold Standard for Partitioning

> A partitioned system should provide fault containment equivalent to an idealized system in which each partition is allocated an independent processor and associated peripherals, and all inter-partition communications are carried on dedicated lines.

Although this Gold Standard provides a suitable mental benchmark for designers and certifiers of partitioning mechanisms for IMA, it is less useful as a "contract" with the "customers" of such mechanisms. These customers—that is, those who develop software for the functions that will run in the partitions of an IMA architecture—are assured that their software will be as well protected in a partition as if it had its own dedicated system, but they are not provided with a concrete environment in which to develop, test, and certify that software. The Gold Standard implies that the environment provided by the partitioned system to a particular application function must be indistinguishable from an idealized system dedicated to that function alone, but this idealized system is just that—an imaginary artifact—and not one suitable for testing and evaluating real-world software. The only environment actually available is the partitioned system itself, so its customers need a contract expressed in terms of that environment. This can be done as follows:[4] instead of comparing the environment perceived by the software in a partition to that of an idealized, dedicated system, we require that the environment (whatever it is) is one that is totally unaffected by the behavior of software in other partitions. This leads to the following alternative statement of our Gold Standard.

Alternative Gold Standard for Partitioning

> The behavior and performance of software in one partition must be unaffected by the software in other partitions.

This formulation is not only simpler and more direct than that involving an idealized system, but it also suggests how the customers of a partitioned system can develop and evaluate their software—for if software in one partition is unaffected by that in other partitions, it will run the same (in terms of both behavior and performance) whether the other partitions are inhabited or empty. Thus, in particular, individual software functions can be developed and certified using the real environment of the partitioned system, but with the other partitions empty (or, more likely, containing stubs to provide the data sources and sinks required by the function under examination). The Alternative Gold Standard ensures that

[4] I am grateful to David Hardin, Dave Greve, and Matt Wilding of Collins Commercial Avionics for explaining this approach and its motivation to me [111].

the certified software will behave exactly the same when those other partitions are inhabited by real (and possibly faulty) functions.

A problem with the Alternative Gold Standard is apparent in the mention of "data sources and sinks" in the discussion above: software functions residing in separate partitions are seldom completely independent—some provide data or control inputs to others. This means that "unaffected by the software in other partitions" needs to be qualified in some way that allows the effects of intended communications while excluding those that are unintended. Thus, although the Alternative Gold Standard is more attractive than the original one as a requirements definition for partitioning *isolated* functions, it needs further development before it can serve as a gold standard for the more general case of partitioned but interacting functions. When restricted to isolated functions, the basic and the Alternative Gold Standards are very similar; indeed, if suitably formalized, each would be definable in terms of the other.

The original formulation of the Gold Standard has the advantage that it focuses attention on the structural differences between a partitioned system and a federated one. These structural differences introduce two classes of hazards into a partitioned system: a fault in one partition could corrupt code, control signals, or data (in memory or in transit) belonging to another, or it could affect the ability of another partition to obtain access to, or service from, a shared resource (such as the processor or a bus). In considering issues in the design and assurance of partitioned systems, it is therefore useful to distinguish two dimensions—spatial and temporal—corresponding to these two classes of hazards.

Spatial Partitioning

> Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit), nor command the private devices or actuators of other partitions.

Temporal Partitioning

> Temporal partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter, and duration of scheduled access to it.

The mechanisms of partitioning must block the spatial and temporal pathways for fault propagation by interposing themselves between avionics software functions and the shared resources that they use. In this way, the partitioning mechanisms

can control or "mediate" access to shared resources. In the next chapter, we consider the mechanisms that can be used to provide mediation in each of the two dimensions of partitioning.

Chapter 3

Issues and Mechanisms

As discussed in the previous chapter, issues in partitioning arise at two levels: within a single processor, and across a distributed system. Issues in partitioning also interact with those in fault tolerance. We consider these topics separately below, and further separate them into consideration of spatial and temporal partitioning.

3.1 Partitioning Within a Single Processor

We start by considering partitioning within a single processor. We sometimes use the neutral term *application* to refer to the computational entity within each partition; this could be a complete avionics function (e.g., a yaw damper), or a part of one. Depending on the implementation, an application could correspond to the operating system notions of *process* or *virtual machine*, or it could be some different notion. An application will generally be composed of smaller units of computation that are called or scheduled separately; we generally refer to these as *tasks*. Again depending on the implementation, these may correspond to an operating system notion such as *thread* or *lightweight process*. Partitioning must prevent applications interfering with one another, but the tasks within a single application are not protected from each other. We focus first on partitioning in the spatial dimension.

3.1.1 Spatial Partitioning

The basic concern of spatial partitioning is the possibility that software in one partition might write into the memory of another: memory is often pictured as a one- or two-dimensional grid, hence the reference to the spatial dimension for this aspect of partitioning. Memory includes that used to store programs as well as data, although in embedded systems it is sometimes possible to hold the former in ROM, where it cannot be overwritten by errant software.

Hardware mediation provided by a memory management unit (MMU) is the usual way to guard against violations of spatial partitioning. The details vary from one processor design to another, but the basic idea is that the processor has (at least) two modes of operation and, when it is in "user" mode, all accesses to memory addresses are either checked or translated using tables held in the MMU. A layer of operating system software (generally called the *kernel*) manages the MMU tables so that the memory locations that can be read and written in each partition are disjoint (apart, possibly, from certain locations used for inter-partition communications). The kernel also uses the MMU to protect itself from being modified by software in its client partitions, and must be careful to manage the user/supervisor mode distinctions of the processor correctly to ensure that the mediation provided by the MMU cannot be bypassed. (In particular, entry and exit from the kernel needs to be handled carefully so that software in a partition cannot gain supervisor mode; some processors have had design flaws that make this especially difficult [44].)

Software executing in a partition accesses processor registers such as accumulators and index registers as well as memory. Generally, the kernel arranges things so that the software in one partition executes for a while, then another partition is given control, and so on; when one partition is suspended and another started, the kernel first saves the contents of all the processor registers in memory locations dedicated to the partition being suspended, and then reloads the registers (including those in the MMU that determine which memory locations are accessible) with values saved for the partition that executes next. The software in the partition resumes where it left off and cannot tell (apart from the passage of time while it was suspended) that it is sharing the processor with other partitions.

The description just given resembles classical time-sharing, where partitions can be suspended at arbitrary points and resumed later. Some variations are possible for embedded systems. For example, if partitions are guaranteed an uninterruptible time slice of known duration, they can be expected to have finished their tasks before being suspended and can then be restarted in some standard state, rather than resumed where they left off. This eliminates the cost of saving the processor registers when a partition is suspended (but at least some of them—including the program counter—must be restored to standard values when the partition is restarted). We can refer to the two types of partition swapping arrangements as the *restoration* and *restart* models, respectively.

In either case, the requirement on the mediation mechanisms managed by the kernel is that the behavior perceived across a suspension by the software in each partition is predictable without reference to anything external to the partition. In the "restoration" model, the processor state must be restored to exactly what it was before suspension; in the "restart" model, it must be restored to some known state. It may be acceptable in the latter case to specify that some registers may be "dirty" on restart and that the software in a partition is required to work correctly without

making assumptions on their initial contents—this saves the cost of restoring these registers to standard values (obviously, the program counter and MMU registers must be restored).[1] The requirement to make behavior predictable across the suspension and resumption of a partition generates in turn the requirement that the operation of the processor must be specified precisely and accurately with respect to all of its registers—for it is important that register saving and restoration or reinitialization should not overlook visible minor registers such as condition codes and floating point/multimedia modes, and that hidden registers, such as those associated with pipelines and caches, really are hidden. (Again, processors often have design glitches, or errors and omission in documentation, that make it difficult to accomplish this [98].)

In the approach just outlined, the mechanisms of spatial partitioning comprise the processor and its MMU, and the kernel. There is much advantage, from the point of view of assurance and formal specification, if these mechanisms are simple. Unfortunately, commodity processors, their MMUs, and associated features such as memory caches, are generally designed for high performance and extensive functionality rather than simplicity. Although a fast processor is often desired, the functionality of MMUs and cache controllers generally exceeds that required for embedded systems; MMUs, in particular, are usually designed to provide a flexible virtual memory and contain large associative lookup tables—whereas for partitioning, a simple fixed memory allocation scheme would be adequate.[2] The latter would also be far less vulnerable to bit-flips caused by single-event upsets (SEUs) than a traditional million-transistor MMU. However, because they are usually highly integrated with their processor, it can be difficult or even impossible to replace MMUs and cache controllers with simpler ones, but consideration should be given to this issue during hardware selection.

An alternative to spatial partitioning using hardware mediation is Software Fault Isolation (SFI) [107]. The idea here is similar to array bounds checking in high-level programming languages, except that it is applied to all memory references, not just those that index into arrays. By examining the machine code of the software in a partition, it is possible to determine the destinations of some memory references and

[1] Although partitioning has much in common with computer security, this is one aspect where they differ: "dirty" registers are anathema in computer security because they provide a channel for information flow from one partition to its successor. The issues underlying this difference are considered on page 75 in Chapter 4.

[2] MMUs are also heavily optimized for speed: in some architectures, the MMU will start a read from the memory using the current page map before it has determined whether that is still valid; if it is not valid, the MMU squashes the bus read transaction before it completes. Also, for efficiency, multiple copies may be maintained for some of the associative lookup tables, and these must be kept consistent with each other. All this is done in the context of speculative out-of-order execution, where providing assurance for correctness of these optimizations is nontrivial. A separate problem is the timing uncertainty introduced by these optimizations: ratios of 2 to 1 between average-case and worst-case timings are not uncommon [52] (see also `http://www.intelligentfirm.com/`).

jumps and hence to check, statically, whether they are safe. Memory references that indirect through a register cannot be checked statically, so instructions are added to the program to check the contents of the register at runtime, immediately prior to its use. By using more elaborate static analysis or program verification techniques (e.g., to ensure that an index register has not been changed since last checked), it is possible to minimize the number of runtime checks; by using modest optimizations of this kind, an overhead of just 4% has been reported for the runtime checks of SFI [107].

Static (i.e., compile-time) analysis of information flow within individual programs written in high-level languages has long been a topic in computer security. In its simplest form, some of the variables used by the program are labeled HIGH and some LOW, and the goal is to check whether information from a HIGH variable can ever influence the final value of one labeled LOW. Techniques for information flow analysis include approximate methods similar to typechecking [21, 106] or to data flow analysis [6], as well as exact methods [63] and those that rely on formal proof [81]. It is possible that approaches based on these techniques could reduce, or even eliminate, the runtime overhead of SFI.

Although SFI usually imposes a small overhead on memory references within a partition, it can greatly reduce the cost of controlled references or procedure calls across partitions (compared with hardware mediation, since the cost of a partition swap is avoided). However, for reasons discussed later (page 49), such cross-partition references may not be acceptable in some partitioned architectures, so the advantage would be moot in those cases.

A disadvantage of SFI compared with hardware-mediated partitioning is that it imposes an additional analysis and certification cost on every program, whereas hardware mediation has the one-time cost of designing, implementing, and certifying the partitioning mechanisms of the kernel and its supporting hardware. On the other hand, the analysis required for SFI lends itself to powerful automation (cf. "extended static checking" [22], and "proof carrying code" [80]) where the certification cost would be transferred to the one-time cost of certifying the tools.

Even without automation, SFI may have advantages of cost and simplicity in "asymmetric" applications where a single function is allocated to a processor but it is desired to include some less critical "nice-to-have" features. These could be partitioned from the main safety-critical function by SFI, while the latter runs unchanged. SFI might also be cost-effective in partitioning functions of similar assurance levels that already require significant analysis (e.g., two Level A functions). And SFI could also be used to provide additional protection *within* partitions (i.e., among tasks) established by hardware mediation.

One concern about SFI, especially when static analysis is used to optimize away many of the runtime checks, is that it provides little protection against hardware faults (e.g., SEU-induced bit-flips) that cause memory addresses that were correct

when analyzed to be turned into ones that are incorrect when executed. The bad memory reference will be caught only if a runtime check is in the right place; a hardware MMU, on the other hand, mediates *every* reference at its time of execution. It was earlier stated that the purpose of partitioning is to protect functions against faults of design and implementation in other functions, not to guard against hardware faults—since these could afflict the function even if it had its own dedicated processor—but a hardware fault that leads to a violation of partitioning is not a fault that would have afflicted the function if it had its own processor, so it seems that the concern is legitimate. However, a little analysis shows that the increased exposure to hardware faults is small. Suppose the function in which we are interested shares its processor with n other functions of similar size, and that the probability of an SEU hitting any one of them is p. Suppose further that the probability that an SEU in one function will cause it to violate SFI partitioning and to afflict some other function is q. Then the probability of an SEU directly or indirectly affecting the original function changes from p to $(1+q)p$ when the function is moved from a dedicated to a shared processor. (Notice that this is independent of n: the chance of an SEU hitting *somewhere* increases by a factor of n, but the chance that the consequent memory error affects the function concerned is reduced by the same factor.) This small increase in probability is unlikely to be significant, and we conclude that the possibility of SEU-induced addressing errors does not invalidate SFI.

Perhaps surprisingly, it is some implementations of hardware-mediated partitioning that seem more vulnerable to this kind of fault scenario. Although an SEU in an individual function cannot lead to a violation of partitioning when memory references are mediated by an MMU, an SEU in the MMU itself could be quite dangerous. If the MMU is a large device with millions of transistors, then the possibility of an upset cannot be overlooked, and a change to one bit in an address translation register may cause the memory references of one partition systematically to infringe on the memory of another. It seems to me that in designs where it is possible to provide a custom MMU, it would be prudent to ensure that this is either fault tolerant, or that it merely checks rather than translates addresses (so that a double fault would be needed to violate partitioning); best of all might be relocation or checking with hardwired values.

So far, our consideration of partitioning has considered only the processor and the memory, and has assumed that different partitions are meant to be isolated from each other; we now need to consider inter-partition communications, and devices. Like partitioning itself, there are two dimensions to inter-partition communication: the spatial dimension is concerned with where and how data is transferred from one partition to another, while the temporal dimension is concerned with whether and how synchronization is performed, and how one partition invokes services from

another. We postpone consideration of the latter topics to the discussion of temporal partitioning in Section 3.1.2 and focus here on the spatial dimension.

The obvious way to communicate data from one partition to another is to copy it from a buffer in memory belonging to the first partition into a separate buffer in the memory of the second. Because only the kernel has access to the memory of both partitions, it must perform the copying and, since it generally runs without memory protection, it must check carefully against buffer overruns. A more efficient scheme uses a single buffer in memory locations that are among those the sending partition can write and the receiver can read (both MMU and SFI forms of memory protection can do this); data can then be copied into the shared buffer by the sending partition without the active participation of the kernel. The receiving partition must assume that the sending one can write arbitrary data anywhere in their shared buffers whenever it has control, and its verification must be performed under this assumption. It seems cleanest if separate buffers are used for each direction of transfer, but bidirectional buffers may also be acceptable. It is, however, important that separate buffers are used for each pair of partitions (otherwise, partition A could overwrite the data of B in C's single input buffer).

Observe that it is important to restrict inter-partition communications to those that are intended: one partition should be able to send data to another only if that communication is authorized in the specification of the system configuration (and the receiving partition must then have a buffer to receive it). A related topic is how one partition should name the other partitions with which it communicates. Absolute addresses (e.g., "send this datum to Partition 7") lead to a rigid and fragile system organization and are to be deprecated on this account. Functional addresses (e.g., "send this datum to the pitch autopilot") are little better: they build assumptions about the system structure into individual applications and limit the opportunities for reuse and reconfiguration. Relative addressing (e.g., "send this datum out on my Port 7") allows the binding of names to specific inter-partition communication channels to be postponed until system configuration time (and may allow some dynamic reconfiguration), but requires a database to record what type of data or service is provided (or expected) on a given port. The best arrangement may be one where partitions use the type of data or service provided or expected as the name of the port concerned (e.g., "send this datum out on my `air-data-samples` port," or "get me an `air-data-sample`"); the binding of these names to inter-partition channels can be done during system configuration, or at runtime. In the latter case, we have something like a publish-subscribe architecture [82]; this provides excellent support for dynamic reconfiguration, but its application to life-critical systems is still an issue for research. (Some avionics systems use this type of naming or addressing scheme, but not in a way that is tightly integrated with their fault-tolerance mechanisms.)

Software in one partition should not make assumptions about when tasks in other partitions are scheduled (tasks within some partitions may be dynamically scheduled); this, combined with normally asynchronous communication, means that care is needed when communicating time-sensitive data. For example, a task that collects from its input buffer a sensor sample contributed by another partition needs to know when that sample was taken. The usual arrangement is to attach a time-stamp to the sample (since both partitions are running in the same processor, they have access to a common clock). However, the utility and interpretation of a sensor sample depends not only on its age, but also on its accuracy and the dynamics of the physical process being measured (e.g., an altimeter reading that is 1 second old is much less useful if the aircraft is landing than if it is in cruise). Some of these factors are likely to be much better known to the partition that provides the sensor sample than to the one that receives it, and duplicating the knowledge in both places is expensive and raises the problem of ensuring consistency. Instead, it seems best if the provider of the data also provides a compact description of its temporal interpretation. Kopetz has made an interesting proposal of this kind under the name *temporal firewall* [53,57], which exists in two variants. A *phase-insensitive* sensor sample is provided with a time and a guarantee that the sampled value is accurate (with respect to a specification published by the partition that provides it) *until* the indicated time. For example, suppose that engine oil temperature may change by at most 1% of its range per second, that its sensor is completely accurate, and that the data is to be guaranteed to 0.5%. Then the sensor sample will be provided with a time 500 ms ahead of the instant when it was sampled, and the receiver will know that it is safe to use the sampled value until the indicated time. This is much more useful than a timestamp that merely records when the sample was taken. A *phase-sensitive* temporal firewall is used for rapidly changing processes where state estimation is required; in addition to sensor sample and time, it provides the parameters needed to perform state estimation. For example, along with the sampled altitude it may supply vertical speed, so that altitude may be estimated more accurately at the time of use.

In addition to communications between partitions, we must examine communications between partitions and devices. Devices, which include sensors and actuators as well as peripherals such as mass storage, have implications for both temporal and spatial partitioning. Most devices raise an interrupt when data is available, or when they need service. Such interrupts affect the timing and locus of control, and consideration of their impact is postponed to the discussion on temporal partitioning in Section 3.1.2; here we concentrate on the relationship of devices to spatial partitioning. Devices impact spatial partitioning in three ways: they need to be protected against access by the wrong partition, they must not be allowed to become agents for violating partitioning, and they may themselves need to be partitioned.

The simplest case is where a device "belongs" to some partition and should not be accessed by others. Most modern processors use memory-mapped I/O, meaning that interaction with devices is conducted by reading and writing to registers that are referenced like ordinary memory locations. In these cases, the mechanisms (MMU or SFI) used to provide ordinary memory protection can also protect devices. If memory protection is insufficiently fine-grained to permit devices to be allocated to partitions as desired, then it will be necessary to create special device management partitions that own several devices but are trusted to keep them separate. Similar arrangements will be necessary if several devices are attached to a data bus or remote data concentrator (and may also be useful if multicast communication services are desired). Of course, the trust in such "multiplexing" partitions needs to be justified by suitable verification and assurance. An alternative to providing device management partitions is to perform these functions in the kernel. The argument against doing this is that the properties of the kernel must be assured to a very high degree, so there is much advantage to keeping its functionality as simple as possible. It should be easier to provide assurance for a kernel that provides memory protection, plus separate device management partitions, than for a kernel having both functions.

Some devices may be shared by more than one partition. Such devices come in two forms: those that need protection and those that do not. An example of the former is a sensor that periodically places a sample in a device register. There seems no harm in allowing two partitions both to have read access to the memory location containing that device register. Devices that accept commands are more problematical in that faulty software in one partition may issue commands that render the device inoperable or otherwise unavailable to other partitions. Protection by a special device management partition seems necessary to mediate access in these cases. (The Clementine spacecraft was lost when a software fault caused garbage to be sent over an unmediated bus, where it was interpreted by an attached device as a command to fire all the thrusters without limit.) Notice that such a device management partition must play a more active role in checking or controlling the device than the simple "multiplexing" device management partitions described earlier.

Device management partitions also are necessary to mediate access to truly shared devices such as mass storage. In these cases, it is usual for the device manager to synthesize a service (e.g., a file system) rather than just mediate access to the raw device (e.g., a disk), and to partition the service appropriately (e.g., with a separate "virtual" file system for each client partition). A device manager of this kind poses challenges to assurance that are similar to those of the main memory partitioning mechanism, since flaws could allow one client partition to write into areas intended for another.

Mass storage and other devices that transfer large amounts of data at high speed generally do so by direct memory access (DMA) rather than through memory-mapped device registers (which are limited to a few bytes at a time). Depending on the processor and memory architecture, DMA devices may be able to address memory directly, without the mediation of the MMU. This arrangement has the potential to violate partitioning since faulty software may instruct the device to use a region of memory belonging to some partition other than its own; a fault in the device itself could have a similar effect. A simple solution is to interpose some checking or limiting mechanism into the device's memory address lines (e.g., by cutting or hard-wiring some of them) so that the range of addresses it can generate is restricted to lie within that of the partition that manages it. Another solution is to isolate each DMA device to a private bus with a dual-ported memory bridging the private and main system buses.

3.1.2 Temporal Partitioning

Our context is real-time embedded systems, where correctness requires not only that the right results are produced, but that they are produced at the *right time*. The concern of temporal partitioning is to ensure that activities in one partition do not disturb the timing of events in other partitions.

The most gross concerns are that faulty software in one partition might monopolize the CPU, or that it might crash the system or issue a HALT instruction—effectively denying service to all other partitions. Other scenarios that can cause a partition to fail to relinquish the CPU on time include simple schedule overruns, where particular parameter values cause a computation to take longer than its allotted time, and runaway executions, where a program gets stuck in a loop.

Although their manifestations are in the temporal dimension, system crashes and instructions that halt the CPU are usually prevented by the mechanisms of spatial partitioning. In particular, HALT and other dangerous instructions usually cannot be issued (or, rather, they cause a trap to the kernel) when in user mode. There are reports, however, that some steppings of some commodity processors have untrapped instructions that can halt the CPU, or user-mode instructions that can "hang" when supplied with certain parameters (e.g., see `http://www.x86.org`; also [98] notes 102 bugs reported up to 1995 in various versions and steppings of the Intel 80X86 architecture, while [8] documents a comparable number in later processors). It is important to know these characteristics of the precise stepping of the processor employed (which may require a nondisclosure agreement), but it is difficult to provide a complete solution to such untrapped hardware flaws. Perhaps the best that can be done is to use SFI-like techniques and to scan the machine code of each application and insert runtime checks as necessary to prevent execution of dangerous instructions or parameter values (a purely static check will be inadequate

if parameter values can be constructed or modified—either under program control or by an SEU—at runtime).

The last-ditch escape from a halted or locked-up CPU is a watchdog timer interrupt managed by the kernel. This is not certain to provide recovery, however, unless the basic kernel design is correct: for example, design faults in the Magellan spacecraft led to a runaway execution in which a program sat in a loop that did nothing but reset the watchdog timer [18, pp. 209–221] [26,51],[3] and not all halted or "hung" processors respond to the timer interrupt. Recovery in these dire cases usually depends on a system reset (or cycling the power supply, which causes a reset), which may be invoked either manually or by other components in a distributed fault-tolerant system (which is how Magellan recovered).

Runaway executions in the kernel, lockups, and untrapped halt instructions could all afflict a processor dedicated to a single function, and so their treatment is more in the domain of system-level design verification or fault tolerance than partitioning. Overruns or runaways within a function, however, are genuinely the concern of partitioning and are usually controlled through timer interrupts managed by the kernel: the kernel sets a timer when it gives control to a partition; if the partition does not relinquish control voluntarily before its time is up, the timer interrupt will activate the kernel, which then will then take control away from the overrunning partition and give it to another partition under the same constraints.

Merely taking control away from an overrunning partition does not guarantee that other partitions will be able to proceed, however, for the overrunning partition could be holding some shared device or other resource that is needed by those other partitions. The kernel could break any locks held by the errant partition and forcibly seize the resource, but this may do little good if the resource has been left in an inconsistent state. These considerations reinforce the earlier conclusion that devices and other resources cannot be directly shared across partitions. Instead, a management partition must own the resource and must manage it in such a way that behavior by one client partition cannot affect the service received by another.

Another problem can arise if the overrunning partition is performing some service on behalf of another partition: it will generally be necessary to notify the invoking partition (the next time it is scheduled) of the failure of the service provided by the other. The invoking partition must have enough fault tolerance that it can do something sensible despite the failure of the service. It may also be necessary for the kernel to perform some remedial action on the partition that overran its

[3]The flaw in Magellan was in the design of its kernel (sensitive data structures were manipulated outside the protection of a critical section, so an interrupt could leave them in an inconsistent state). Such flaws would be unconscionable in a safety-critical system: the design of the core hardware and software mechanisms simply have to be correct in these systems. In addition to skilled and experienced designers, formal methods of specification and analysis may be valuable for this purpose (design diversity is implausible at these levels).

allocation. This could force that partition to do a restart next time it is scheduled, or could simply notify the partition of its failure and leave recovery (e.g., the killing of orphans) to the operating system functions resident in that partition.

Timeout mechanisms such as those just described ensure that each partition will get *enough* access to the CPU and other resources, but real-time systems need more than this: the tasks within partitions need to get access to the CPU and to devices and other resources at the *right* time, and with great *predictability*. This means that discussion of temporal partitioning cannot be divorced from consideration of scheduling issues. The real-time tasks within a partition generally consist of *iterative* tasks that must be run at some fixed frequency (e.g., 20 times a second) and *sporadic* tasks that run in response to some event (e.g., when the pilot presses a button); iterative tasks often require tight bounds on *jitter*, meaning that they must sample sensors or deliver outputs to their actuators at very precise instants (e.g., within a millisecond of their deadline), and sporadic tasks often have tight bounds on *latency*, meaning that they must deliver an output within some short interval of the event that triggered them.

There are two basic ways to schedule a real-time system: statically or dynamically. In a static schedule, a list of tasks is executed cyclically at a fixed rate. Tasks that need to be executed at a faster rate are allocated multiple slots in the task schedule. Even sporadic tasks are scheduled cyclically (to poll for input and process it if present). The maximum execution time of each task is calculated, and sufficient time is allocated within the schedule to allow it to run to completion: thus, one task never interrupts execution of another (although a task may be terminated if it exceeds its allocation). Notice that this means that a long-duration task may need to be broken into several smaller pieces to make room for short tasks with higher iteration rates. The schedule is calculated during system development and is not changed at runtime (although it may be possible to select among a fixed collection of different schedules at runtime according to the current operating mode).

In a dynamic schedule, on the other hand, the choice and timing of which tasks to dispatch is decided at runtime. The usual approach allocates a fixed priority to each task, and the system always runs the highest-priority task that is ready for execution. If a high-priority task becomes ready (e.g., due to a timer or external interrupt) while a lower-priority task is running, the lower-priority task is interrupted and the high-priority task is allowed to run. Note that this requires a context-switching mechanism to save and later restore the state of the interrupted task. The challenge in dynamic scheduling is to allocate priorities to tasks in such a way that overall system behavior is predictable and all deadlines are satisfied. Originally, various plausible and ad-hoc schemes were tried (such as allocating priorities on the basis of "importance"), but the field is now dominated by the rate monotonic scheduling (RMS) scheme of Liu and Layland [66]. Under RMS, priorities are simply allocated on the basis of iteration rate (the highest priorities going to the tasks with the highest

rates) and, under certain simplifying assumptions, it can be shown that all tasks will meet their deadlines as long as the utilization of the processor does not exceed 69% (the natural logarithm of 2—higher utilizations are possible when the task iteration rates satisfy certain relationships). Some of the simplifying assumptions (e.g., that the context-switch time is zero, and that tasks do not share resources) have been lifted or reduced recently [62, 69, 96].

The choice between static and dynamic scheduling is a contentious one (Locke [67] provides a good discussion). The basic arguments in favor of static scheduling are its complete predictability and the simplicity of its implementation; the arguments against are that all tasks must run at a multiple of the basic iteration rate (so that some run more or less frequently than is ideal for their control function), the handling of sporadic tasks is wasteful, and long-running tasks must be broken into multiple, separately scheduled pieces (to make room for tasks with faster iteration rates). The arguments in favor of dynamic scheduling are that it is more flexible and copes better with occasional task overruns; the arguments against hinge on the difficulty of giving complete assurance that a given task set will always meet its deadlines under all circumstances. (The factors that must be considered are complex and not all are fully characterized; errors of understanding or judgment are not uncommon. For example, the much publicized communications breakdowns between the 1997 Mars Pathfinder and its Sojourner rover were due to priority inversions in its RMS scheduler.[4] Priority inversions are a well-understood problem in dynamically scheduled systems, with a well-characterized solution called "priority inheritance" [20, 96] that was available, but not used, in the commercial real-time executive used for Pathfinder.)

The mechanisms of both static and dynamic scheduling have to be modified to operate in a partitioned environment, and these modifications change some traditional expectations about the tradeoffs between the two approaches; in addition, partitioning creates opportunities for hybrid approaches that combine elements of both basic mechanisms. The traditional scheduling problem is to ensure satisfaction of all deadlines, given information about the rate and duration of the tasks concerned. It is assumed that this information is accurate; if it is not—if, for example, some task runs longer or requests service more often than expected—then the system may fail. When all the tasks in the system are contributing to some single application, such a failure may be undesirable but will not have repercussions beyond those consequent on the failure of the application concerned. In a partitioned system, however, it is necessary to ensure that faulty assumptions about the temporal behavior of tasks belonging to one application cannot affect the temporal behavior of applications in different partitions.

[4] See `http://www.research.microsoft.com/research/os/mbj/Mars_Pathfinder/Authoritative_Account.html`.

There seem to be two ways to achieve this temporal partitioning: one is a two-level structure in which the kernel schedules *partitions*, with the application in each partition then responsible for locally scheduling its own tasks; the other is a single-level structure in which the kernel schedules *tasks*, but with a quota system to limit the consequences of any faults—or faulty assumptions—to the partition that is in violation.

The first approach usually employs static scheduling at the partition level: the kernel guarantees service to each partition for specified durations at a specified frequency (e.g., 20 ms every 100 ms) and the partitions then schedule their tasks within their individual allocations in any way they choose; in particular, partitions may use dynamic scheduling for their own tasks. Any partition that schedules its tasks dynamically must provide a mechanism for interrupting one task in favor of another. Such support for task swapping is one of the reasons for preferring dynamic over static scheduling: it simplifies application programming by allowing long-running, low-frequency tasks to be interrupted by shorter high-frequency tasks, whereas statically scheduled systems have to break long-running tasks into separately scheduled fragments that perform their own saving and restoration of local state data to create room for the higher-frequency tasks. If partition swapping uses the restoration model, however, it provides an alternative mechanism for dealing with long-running tasks within a statically scheduled environment: a single application can be divided into parts that are allocated to separate partitions that are scheduled at different rates. The partition-swapping mechanism then takes care of interrupting and restoring the long-running tasks, thereby simplifying their construction.

Opportunities such as this make static scheduling for both partitions and tasks relatively attractive. Conversely, the constraints of static partition scheduling render its combination with dynamic task scheduling rather less attractive. One of the conveniences of dynamic scheduling is that it allows new tasks to be introduced—or the frequency and duration of existing tasks to be changed—with relative ease. But this ease is vitiated when partitions are statically scheduled because, for example, a new 10-Hz task can only be fitted into a partition that is already scheduled at this rate (or some multiple of it), so that the rigidity of the partition-scheduling mechanism dominates any flexibility in task scheduling.

This drawback could be overcome, however, if partitions could be scheduled at iteration rates very much higher than those of any task—say 1,000 times a second. Under the restoration model of partition swapping, a partition that is scheduled at such a rate and that is guaranteed, say, one tenth of the CPU (i.e., 100 μs every millisecond) could, for most purposes, be regarded as running continuously on a CPU that has one tenth the power of the real one, and its tasks could be dynamically scheduled without regard to the underlying partition schedule. Partition swaps are relatively expensive on traditional processors (because there is a large amount of state information that has to be saved and restored) and this renders kilohertz

partition schedules infeasible on such hardware (all the resources of the system would be expended in swapping). However, specialized processors are under development where partition swapping is performed at the microcode and hardware levels, and these are believed to be capable of supporting partition schedules in the kilohertz range with no more than 5% to 10% of the system resources expended on swapping. Notice that the task swapping required for dynamic scheduling within each partition can be relatively lightweight (since tasks within a partition are not protected from each other) and will be activated at a frequency comparable to the fastest task iteration rate and not the much faster partition swapping going on beneath it.

The radical combination of a static partition schedule operating at kilohertz rates and dynamic task scheduling within each partition is an attractive one: it seems to provide both the convenience of dynamic scheduling and the predictability of static scheduling. However, one of the conveniences of dynamic scheduling is the ease with which it can accommodate aperiodic activities driven by external events, such as operator (e.g., pilot) inputs and device interrupts, and it requires care to support this on top of static partition scheduling—even when this is running at kilohertz rates. The basic concern is that external events of interest to one partition must not disturb the temporal behavior of other partitions. If partitions are scheduled dynamically, use of suitable quota schemes can allow temporal predictability to coexist with aperiodic event-driven task activations (this is discussed on page 50), but static partition scheduling ensures predictability through temporal determinism and this imposes strong restrictions on event-driven activations.

First and most obviously, a static partition schedule does not allow an external event to initiate a partition swap: the partition schedule is driven strictly by the processor's internal clock, so that if an event requires the services of a task in a partition other than the current one, it must wait until the next regularly scheduled activation of the partition concerned. This increases latency, but may not be a problem if partitions are scheduled at kilohertz rates. Less obvious, perhaps, are the consequences of the requirement that the currently executing partition should see no temporal impact from the arrival of events destined for other partitions. Even the cost of a kernel activation to latch an interrupt for delivery to a later partition reduces availability of the CPU to the current partition and must be strictly controlled. It is possible to add padding to the time allocated to each partition to allow for the cost of kernel activity used to latch some predicted number of interrupts for other partitions. But this makes temporal correctness of one partition dependent on the accuracy of information provided by others (i.e., the number and rate of their external events)—and even originally accurate information may become useless if a fault causes some device to generate interrupts constantly.

This concern is a manifestation of a more general issue: temporal partitioning requires not only that each partition has access to the resources of the system at guaranteed intervals, but that those resources provide their expected performance.

A CPU whose performance is degraded by the cost of latching interrupts for later delivery is just one example; others include a memory subsystem degraded by DMA transfers on behalf of other partitions, or an I/O subsystem that is busy on their behalf.

Under static partition scheduling, temporal partitioning is predicated on determinism: because it is difficult to bound the behavior of faulty partitions, the availability and performance of each resource is ensured by guaranteeing that no other partition can initiate *any* activity that will compete with the partition scheduled to access the resource. This means that no CPU or memory cycles may be consumed other than at the behest of the software in the currently scheduled partition. Thus, in particular, there can be no servicing of device interrupts, nor cycle-stealing DMA transfers other than those initiated by the current partition. These requirements can be violated in two ways: a previously scheduled partition may have had some I/O activity pending when it was suspended, or the external environment may generate an interrupt spontaneously (e.g., to indicate that a button has been pressed).

Draconian measures seem necessary to prevent these sources of temporal uncertainty. External events either should not generate interrupts (the relevant partition should poll for the event instead), or it should be possible to defer handling them until the relevant partition is running (whether this is possible depends on the nature of the device and the interrupt, and on how selectively the CPU architecture allows interrupts to be masked off). Similarly, interrupts due to pending I/O from a device commanded by a previous partition should be masked off. If interrupts cannot be masked with sufficient selectivity, we could require the kernel to issue commands that quiet the devices of the previous partition as part of the process of suspending that partition and starting the next. Alternatively, if devices go quiet when uncommanded for some short time, the kernel could make the device registers unavailable (e.g., by changing the MMU table) during the final few milliseconds of each partition's schedule.

The restrictions just described as necessary to ensure that temporal correctness of tasks in one partition are unaffected by software in other partitions have consequences for inter-partition communications. With static scheduling of partitions, a task that needs the services of software in another partition (e.g., to access a shared device) cannot simply issue a procedure call. In fact, there can be no synchronous services (i.e., where the caller blocks and waits for the service provider to reply) across partitions because (a) one partition should not depend on another (that may be faulty) to unblock its progress, and (b) it would impose a large performance penalty: the caller would block at least until its next slot in the schedule after the service provider's slot. Instead, all inter-partition communication must be asynchronous (where the caller places requests in the input buffers of tasks in other partitions and continues execution; when next activated, it looks in its own input buffers for replies, requests, and unsolicited data from other partitions). Because

faulty software could generate an excessive number of requests for service by another partition, it seems necessary that fixed quotas should be imposed on the number or rate of service requests that will be honored from each partition.

Some of the restrictions that are necessary when partitions are scheduled statically may possibly be relaxed when they are scheduled dynamically. It makes little sense to schedule partitions dynamically and tasks statically, and when both partitions and tasks are scheduled dynamically there is little point in maintaining two levels of scheduling, so the unit of scheduling will actually be the task. However, the concern for temporal partitioning will influence which tasks are eligible for execution. Whereas static scheduling ensures temporal partitioning through strict preplanned determinism, dynamic scheduling relies on theorems from the mathematical study of (for example) RMS. There are two problems in applying this theory in the context of partitioning: one is that a faulty partition may violate the assumptions underlying the theorem concerned; the other (related) problem is that the simplest (and therefore, for life-critical applications, preferred) theorems make the strongest assumptions (e.g., that context switches take no time), whereas those with more realistic assumptions rest on more elaborate and less well-established theory. Both problems can probably be overcome by having the kernel and its scheduler enforce quotas.

For example, if schedulability of a task set is predicated on a given partition taking no more than 20% of the available time in each cycle, then the kernel can simply refuse to make any of its tasks eligible for scheduling once that 20% quota has been reached. The problem with this simple scheme is that a faulty partition may consume its quota in very many small bursts (or a device may generate interrupts at a rapid rate). The many partition swaps entailed thereby may have a more deleterious effect on the tasks of other partitions than the CPU time directly consumed by the faulty task. A plausible way to overcome this problem is to subtract the cost of a partition swap (and the performance degradation caused by disturbing the caches) from the quota of the task that causes it. Quotas managed in this way provide many of the guarantees of static scheduling while retaining some of the flexibility of dynamic scheduling. For example, such a scheme could allow synchronous as well as asynchronous inter-partition communications, together with the ability to service aperiodic events and interrupts. (Modern operating systems such as Scout use a somewhat similar approach, in which accounting for resource usage is performed on abstractions called *paths* [102].) However, many of the restrictions and concerns discussed for static partition scheduling remain relevant for dynamic scheduling: for example, it still seems necessary to eliminate cycle-stealing DMA transfers and other performance-degrading activities that cannot easily be controlled by quotas, and it is also necessary to ensure that interrupts for a partition that has exceeded its quota are masked or latched at truly zero cost. Other potential sources of cross-

partition interference such as locks and semaphores must also be suitably controlled (probably by elimination).

Quota-based dynamic scheduling may provide simple guarantees that the tasks of nonfaulty partitions receive their expected allocations (i.e., they receive *enough* time), but guarantees that they will hit their deadlines (i.e., they get it at the *right* time) are more problematical (there are, for example, scenarios under RMS where the *early* completion of one task causes another to miss its deadline [85]). In practice, relatively few tasks may need to be scheduled with great temporal precision: it is generally necessary to sample sensors and control actuators with very low jitter, but it does not greatly matter when the control laws are evaluated provided their results are ready when needed. Thus, we can envisage a scheme in which certain tasks (those associated with sensors and actuators) are guaranteed to execute with great temporal accuracy, while others are guaranteed only to get their allocation of resources sometime during their period. To achieve this, the sensor and actuator tasks could run in separate processors that are statically scheduled (and communicate with the dynamically scheduled computational tasks through dual-ported memory), or they could run at the highest priority in the dynamically scheduled system; justification for the latter scheme would require deeper theorems than the former.

Whether partitions and tasks are statically or dynamically scheduled, the kernel must collaborate with other software to provide some of the services of an operating system—at the very least it will be necessary to service interrupts. Under static partition scheduling, interrupts from external devices are allowed only when their partition is running; this means it is possible to vector interrupts directly to handlers in the partition, rather than handle them in the kernel. The advantage of the former arrangement is that it minimizes the complexity of the kernel; its difficulty is that interrupts are often vectored in supervisor mode, which can threaten hardware-mediated spatial partitioning. Compromise arrangements have the kernel fielding the hardware interrupt, but then passing it in a safe way to the partition for service. Arguments against device handling in a partition are that this really is an operating system service that is better done by an operating system. A conventional operating system is unattractive in a partitioned environment because, as portrayed in Figure 2.1(a) on page 27, it is a large shared resource that must be shown to respect partitioning as well as to be free of other faults. A more suitable arrangement provides operating system services separately within each partition, as portrayed previously in Figure 2.1(b). This arrangement has the additional benefit that different partitions can use different sets of operating system services: for example (see Figure 3.1 on page 52), a critical function might use a minimal set of services (Partition C), while a less critical but more complex function might employ something close to a COTS operating system (Partition B), and a device management partition might consist largely of standardized operating system services for

device management. Operating system services cannot affect basic partitioning in this arrangement; however, they must be used with great circumspection in partitions that encapsulate a shared service or resource (e.g., a partition that provides a shared file system). Such partitions are logically an extension of the kernel and must be shown to partition their service or resource appropriately—which is likely to be more difficult the more software they contain.

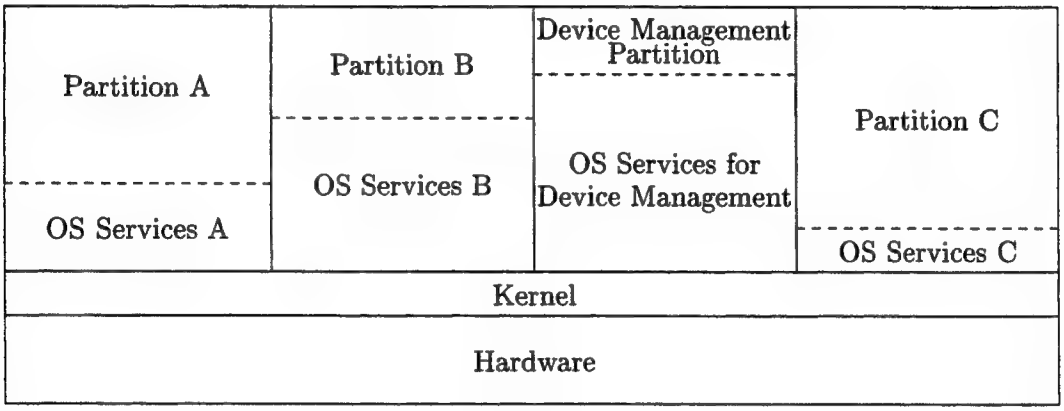


Figure 3.1: Different Operating System Software for Different Partitions

3.2 Partitioning Across a Distributed System

A distributed system resembles our original Gold Standard—a separate processor for each partition—more closely than a single shared processor, and might seem to raise few new issues with respect to partitioning: if we accept that the partitioning mechanisms employed within individual processors are sound, then connecting several such systems together surely cannot do any harm. This would be true if we could arrange dedicated physical point-to-point communications between *partitions* in different processors, but the only physical communications that can be provided are between *processors.* This limitation has a fairly significant impact, which is compounded when we consider shared communications, such as buses.

To start with, suppose we wish to communicate data from partition a_1 of processor A to partition b_1 in a different processor B, and that we have a suitable communications line from A to B. Interrupts will be generated at B as the data starts to arrive and, as we discovered in the previous section, some care is needed to ensure that these do not disturb temporal partitioning in B. If B is dynamically scheduled, the quota schemes discussed previously may be all that is needed, but matters can be more complicated when partitions are scheduled statically. Under

static scheduling, we must require either that the interrupts can be latched at no cost until the scheduled execution of partition b_1 (partitions must be scheduled at high-frequency to make it feasible to service communications in this way), or that partition b_1 (or some device management partition that handles the communications line) is guaranteed to be executing when the interrupts arrive. The latter clearly requires synchronization between the partition schedules of processors A and B and, by extension to other processors, this implies global synchronization of schedules across all processors.

The only way to avoid these consequences when static partition scheduling is employed is to have a data concentrator device at B that buffers incoming data without imposing a load on the CPU or its buses. The partition b_1 can then retrieve incoming data from the data concentrator as part of its normally scheduled activity. A more aggressive design would allow the data concentrator to write incoming data directly into buffers associated with each partition using dual-ported RAM. Even these designs do not necessarily eliminate the need for global synchronization, however, because of the need to control "babbling idiot" failures in partitions and processors.

These are failures where a transmitter sends data constantly, possibly overwhelming its recipient, or denying service to other transmitters. One scenario would be a runaway in partition a_1 that causes it to transmit to b_1 throughout its scheduled execution. We need to be sure that this heavy load on the communications line from A does not affect the ability of the recipient (B or its data concentrator) to service its other lines. This requires either some kind of quota scheme at the recipient, or a global schedule that excludes simultaneous transmissions. A babbling partition can do so only during its scheduled execution, so a global schedule may be able to ensure that no two processors simultaneously schedule partitions that transmit to the same recipient. An alternative if a_1 does not drive the communications line directly, but instead sends data to a device management partition, is for the management partition to impose a quota on the quantity of data that it will accept from any one partition. A babbling processor is an even more serious problem than a babbling partition; either the recipient must be able to tolerate the fault, or it must be prevented at the transmitter—mechanisms to do this are discussed below in the context of bus communications.

The measures discussed above address temporal partitioning in inter-processor communications; we also need to consider spatial partitioning. The spatial dimension to partitioning requires mechanisms to ensure that partition a_1 of processor A can send data to partition b_1 in a different processor B only if that communication is authorized. No additional mechanisms are required to ensure this when a communication line is dedicated to a specific inter-partition channel; additional mechanisms are needed, however, when one line is shared among multiple receiving partitions. In this case, the address of the intended recipient must be indicated

in each transmission. This can be done explicitly by including the address in the data transmitted, or implicitly through the time at which it is sent (the schedules of the sending and receiving processors must be coordinated in this case). A concern with explicit addresses is that a communications fault could transform a datum addressed to partition b_1 into one addressed to b_2. This is a fault-tolerance issue, and is generally handled by checksums or similar techniques to ensure the integrity of transmitted data. The related partitioning issue is the concern that a fault in the sending partition a_1 could cause it to address data directly to an unauthorized recipient b_2—this fault will not be detected by checksums, since it occurs outside their protection. The only certain way to contain this fault is to mediate the communication with some trusted entity that has independent knowledge of the authorized inter-partition communications. This can be performed either at the transmitter (e.g., if a device management partition is used to access the communications line) or at the receiver (e.g., in a data concentrator). A probabilistic method to contain the fault is to allocate partition addresses randomly from a very large space; the chance that a fault in a_1 will cause it to manufacture the legitimate address b_2 is then correspondingly small. In the case of implicit addresses, the concern is that by sending data at the wrong time, the transmitting partition will cause it to be received by an unintended recipient. Mediation is required to contain this fault, which is considered in more detail below, in the context of bus communications.

Some architectures allow the components of a distributed system to communicate without adding explicit addresses to name the intended recipient. In "publish-subscribe" architectures [82], for example, data is tagged with a description of its *content* (e.g., `air-data-samples`) and recipients "subscribe" to data carrying given tags. These issues of naming and binding were discussed earlier (page 40) in the context of individual processors, and similar considerations apply here, but with the added concern for fault tolerance with respect to communications faults.

Using separate communications lines to connect each pair of processors is expensive, so buses are generally used in practice. A bus is a departure from the Gold Standard—it is a resource shared by all processors and all partitions—and it is therefore crucial to provide partitioning so that a fault in one partition or processor cannot affect others. The faults of greatest concern with buses are those where a partition or processor either babbles or fails to follow the access protocol in some way, so that other partitions or processors are denied timely access to the bus.

A babbling or misbehaving partition cannot interfere with bus access by other partitions in its own processor (because a partition can access the bus only when it is scheduled), but it can interfere with access by other processors (by contending for the bus if this is mediated, or by sending transmissions that collide with those of other processors if it is not), and it may overwhelm its receivers. A babbling or misbehaving processor is even more disruptive than a babbling partition because it is not constrained by its own schedule and can monopolize the bus. Notice that

processor faults such as this are partitioning—not fault-tolerance—issues, because their consequences would not be so serious if the buses were not shared. Dual or multiple buses can be used, in the hope that a babbler will confine itself to just one of them, but this cannot be guaranteed. The only certain way to prevent babbling is to mediate each processor's access to the bus by some component that will fail independently. The question then is how does the mediator know what is a legitimate transmission and what is babbling? The answer depends on whether communications are *time* or *event triggered.*

In a time-triggered system, transmissions are determined by a schedule, and the mediating component need only have an independent copy of its processor's schedule and an independent clock in order to determine whether its processor should be allowed to transmit on the bus. The schedules that govern time-triggered transmissions can be either local or global. A local schedule treats each processor independently, so that different processors may contend for the bus and the receiving partition need not be scheduled at the same time as the transmitter. A global schedule, on the other hand, coordinates *all* processor and bus activity, so that there is no bus contention. Although it is perfectly feasible to use global scheduling with contention buses such as Ethernet or CAN (global synchronization means that their ability to resolve contention will never be exercised, but the system benefits from the low cost and high performance of the network interface hardware), some specialized buses have been developed specifically to support and exploit static global scheduling. Examples include the ARINC 659 SAFEbus™ [2, 42] and the Time Triggered Protocol and its associated Architecture (TTP/TTA) [58]. With global scheduling, there is no real need to include a destination address with the data (because this is implicit in the time the message is sent) and some globally scheduled buses (e.g., ARINC 659) do eliminate explicit addresses, thereby reducing the number of bits that need to be communicated and increasing the useful capacity of the bus.

The clock of a bus mediation component needs to be independent of that of its processor, but synchronized with it. With local scheduling, the purpose of the mediating component is to control the pacing of bus accesses, but not their absolute timing and for this purpose it is adequate for the mediator and its processor to synchronize locally (obviously, this must be done carefully to maintain plausibility of the independent failure assumption). With global scheduling, however, the clocks of all processors and mediators must be globally synchronized, and the mediating components should perform the synchronization independently of their processors. If clock synchronization is achieved by a high-level protocol, then the mediating components must be capable of interpreting the full protocol hierarchy, and this greatly complicates their design. For this reason, the mediating components in TTA (called *bus guardians*) do not perform independent clock synchronization, but take synchronizing signals from their host processors [103]. This design prevents babbling, but a processor that loses clock synchronization will take its bus guardian

with it and will still be able to access the bus at the wrong time, though only for short periods. However, the unsynchronized processor/guardian pair will also be unable to receive messages correctly (because synchronization is required to satisfy the CRC checks on each message), and the guardian will shut off all bus access after failing to receive a set number of expected messages. An alternative approach performs clock synchronization as a low-level protocol that can be performed by simple mediating components. This approach seems to require suitable electrical properties of the bus and its drivers. In SAFEbus, for example, the signals from separate drivers are OR'ed together on the bus, and this allows a very simple synchronization protocol that is performed directly in the mediating components (they are called Bus Interface Units in SAFEbus) [2].

Whereas globally scheduled systems guarantee that the bus will be free when a processor is scheduled to transmit, locally scheduled and event-triggered systems must cope with contention between processors attempting to transmit on the bus. In buses intended for control applications, contention is not resolved probabilistically following collisions as it is in classic Ethernet, but deterministically using preassigned slots (as in Echelon's LON), a circulating token (as in PROFIBUS [Process Field Bus] [23]), or a priority arbitration scheme (as in CAN [Controller Area Network] [47]) to provide distributed mutual exclusion and thereby prevent collisions. This determinism does not provide very strong guarantees on how long a processor must wait to access the bus, however. In CAN, for example, a processor that wishes to transmit must first wait for any current transmission to finish and then it must contend with any other processors that also wish to transmit. In CAN, the lowest-numbered processor always wins the arbitration and may therefore have to wait only as long as the longest message transmission, while other processors also have to wait while any lower-numbered processors perform their transmissions.[5] It follows that only probabilistic guarantees can be given on the bus-access delay in such systems, and that these guarantees will be quite weak in the presence of faults [105], even if bus access is mediated to control the worst manifestations of babbling.

It is not straightforward to mediate a processor's access to the bus when that access is event triggered—that is to say, triggered by the processor's internal computations, possibly based on data it has received—for there is no way to know whether an event has legitimately occurred without independently copying the data received and reproducing the computation performed by that processor. A master-checker

[5]The Echelon LON protocol has similar characteristics: stochastic flow control is used to reduce the likelihood of collisions; if a collision does occur, processors back off and access the bus in order of their "contention slots." The main application of the LON protocol is in automating buildings, where tight real-time guarantees are unlikely to be required, but the Echelon web site `http://www.lonworks.echelon.com` reports that Raytheon uses this technology in its Control-By-Light™ fault-tolerant fiber optic distributed control system, which is currently undergoing FAA Part 25 certification for use in commercial aircraft; however, it seems that mechanisms in addition to the LON protocol are employed in this application.

dual-processor arrangement such as this is a very expensive way to prevent babbling. Redundant processors are obviously required for fault tolerance in IMA, but such redundancy should be managed flexibly at the system level, not committed to pairing. Without master-checker pairs, the best that can be done to control babbling in an event-triggered system seems to be the imposition of some limit on the rate at which a processor may transmit on the bus. The ARINC 629 avionics data bus [3] has this capability (the bus uses time slots to control access, but it can be used in the larger context of an event-triggered system).

Because the purpose of partitioning is to control fault propagation, some aspects of partitioning are very close to fault tolerance—for example, the control of babbling discussed in the previous paragraphs has elements of both. Mechanisms such as these are needed to preserve the integrity of the service provided by an IMA architecture to the avionics functions that it supports. In addition, the avionics functions often need to be fault tolerant themselves, and an IMA architecture must therefore support the development of such fault-tolerant applications. There is a choice in how much fault tolerance should be provided by the IMA architecture, and how much by the functions themselves. Faults such as babbling, which are outside the control of any single function and that can have system-wide ramifications, must clearly be tolerated by mechanisms of the IMA architecture. Sensor failure, on the other hand, seems more naturally the responsibility of the function that uses the sensor, while failure of a processor seems to fall somewhere in between—the designers of the function may best know how to handle such a fault, but may need services provided by the IMA architecture to implement their strategy.

As mentioned in Section 2.1 (page 28), the trend toward IMA runs in parallel with another trend toward developing avionics functions on top of a layer that provides standard operating system services and, possibly, additional services to support systematic fault tolerance. Fault tolerance in critical systems is usually based on active redundancy; errors are detected or masked through comparison or voting of the redundantly computed values. Fault tolerant architectures differ in whether the redundant replicas perform the same or different computations, and in whether their states are synchronized (to allow exact-match voting). Some of the architectural choices for fault tolerance are strongly contingent on other choices—for example, that between time- and event-triggered architectures—that are themselves strongly tied to choices in partitioning mechanisms. Kopetz presents persuasive arguments that time-triggered architectures are the best choice for critical real-time applications [54–56] and this choice also fits well with the requirements and mechanisms, discussed in the previous section, for ensuring temporal partitioning in a distributed system.

3.3 Summary

The topics examined in this chapter show that partitioning interacts rather strongly with several other issues in system design: for example, scheduling, communication, distribution, and fault tolerance. By "interacts with" I mean that design freedom in these dimensions is curtailed when partitioning is a primary system goal. This is not necessarily a bad thing, however, because the restrictions imposed by partitioning are exactly those that prevent unexpected interactions among system components, thereby promoting compositionality (i.e., the property that components that work on their own continue to do so when combined with other components) and reducing integration costs.

Because partitioning is critical to the safe deployment of IMA, the design and implementation of its mechanisms must be assured to very high standards. Guidelines for the assurance and certification of safety-critical airborne software are specified in the document known as DO-178B in the USA and ED-12B in Europe [84]. These guidelines call for a very rigorous—if traditional—process of reviews, analysis, and documentation; however, an appendix includes *formal methods* among the "alternative methods" that "may be used in satisfying one or more of the objectives" described in the document. The idea behind formal methods is to construct a mathematical model of a software or system design so that calculations based on the model can be used to predict properties of the actual system—in much the way that finite element analysis of a structural model for an airplane wing can be used to predict mechanical properties of the actual wing. Because the appropriate mathematical domain for modeling software is mathematical logic, where "calculation" is performed by so-called "formal deduction" (as opposed to, say aerodynamics, where the appropriate mathematical domain is partial differential equations, and calculation is performed by numerical methods), this approach is referred to as use of "formal methods."

The utility of calculation—as an adjunct to, or replacement for, physical experimentation—is well understood in other branches of engineering, and is similar in computer science. In fact, its utility is potentially greater in computer science than in other engineering disciplines because computer science deals with discrete or discontinuous phenomena, where experimentation and testing are of limited value as assurance methods. With discontinuous systems, there may be little relationship between the behavior of the system in one circumstance and its behavior in another "similar" circumstance; consequently, extrapolation from tested to untested cases is of doubtful validity. This contrasts with physical systems, where continuity justifies safe extrapolation from limited test cases. Formal methods augment testing by allowing *all* the behaviors of a system to be examined. Formal methods consider a *model* of the system, whereas testing examines the real thing, so the two approaches complement each other. An elementary description of formal methods,

and their application to the certification of avionics is presented in [92], with more detail available in [91].

In addition to their role in assurance, the models constructed in formal methods can often help clarify requirements and design choices, and can lead to improved understanding of design problems. They do this by abstracting away all detail considered irrelevant to the problem at hand, and by formulating the remaining issues with mathematical precision. Formal models for partitioning could therefore help refine our understanding of this topic. Now, partitioning has much in common with certain issues in computer security, and those issues have been the target of considerable research in formal modeling extending over more than two decades. The next chapter, therefore, examines issues in computer security related to partitioning, and outlines the formal modeling techniques that have been tried.

Chapter 4

Comparison With Computer Security

Computer security is related to partitioning in that both are concerned with the ability of one software application to influence another. The concerns are that sensitive information might "leak" from one partition to another (this is called *information flow* in the security context), or that doubtful information might contaminate high-quality information (this is called information *integrity* in the security context), or that one partition might monopolize or reduce timely access to the CPU or some other resource (this is called *denial of service* in the security context). Much work over many years (see, for example, a survey published in 1981 [61]) has sought to provide a firm understanding of these security issues and their enforcement mechanisms, and we might hope to apply some of this work—or at least the underlying ideas—to partitioning. In addition, research in computer security has sought to provide rigorous, formal approaches to the specification and verification of secure systems, and there is hope that these approaches could contribute to the development of strong assurance techniques for partitioning in avionics. The following sections review these security issues and the formal modeling techniques that have been applied to them. The goal here is to explain the basic ideas and approaches, so we merely describe the formal techniques that have been used rather than present the actual formalism.

4.1 Data and Information Flow

The most studied aspect of computer security is something of a dual to one of the concerns of spatial partitioning. In spatial partitioning, a concern is that one partition might write data into a second, and thereby disrupt its operation. In security we are more concerned with the data that is written: if data in the first

partition is considered highly classified, then writing it into a more lowly classified partition is tantamount to disclosing it. Reflecting this concern, computer security generally uses the more neutral term *process* for what was called a partition in the previous chapter (indeed, the computer security notion that is closest to partitioning is called "process security" [9]). *Data flow* security is concerned with controlling channels for disclosure; *information flow* security is concerned with more subtle channels in which data is not written directly, but its information content is disclosed just as effectively.

4.1.1 Access Control

A basic mechanism in enforcing both partitioning and information flow security is called *access control*: the computer system is assumed to have some means (typically, supervisor/user mode distinctions and memory management hardware) for limiting the primitive resources that a process can access, and the ways in which it can access them. Then some higher-level resources are synthesized (e.g., a file system), and rules governing access to those resources are defined and implemented in terms of the more primitive resources and protections. The rules constitute an *access control policy*. A familiar example is that of the Unix file system: each file is associated with a particular owner and group, and we can specify separately whether the owner, members of the group, or other users can read, write, or execute the file. This example raises two important topics in access control: the first concerns the choice and specification of the access control policy that is to be enforced, and the second concerns the completeness of that enforcement.

The Unix file system provides a *discretionary* access control policy: users who have read access to a file can, at their discretion, copy it and grant access to the copy in any way they choose. This may be contrary to the intent of the original owner, or to some organizational policy. To deal with these concerns, various more constrained kinds of *mandatory* access control policies have been defined. The simplest example is the *multilevel* security policy that is intended to reflect practices for handling classified military information. In a multilevel policy, every resource and every process (computer security uses the terms *object* and *subject* for these) is given a label from some ordered set (typically UNCLASSIFIED, CONFIDENTIAL, SECRET, and TOP SECRET), and a subject may have read access to an object only if the subject's label (its *clearance*) is equal to or greater than that of the object (its *classification*).[1] This rule (it is called the *simple security property*) does not stop a subject creating a copy of an object at a lower classification and thereby violating the intent of the policy, so it is augmented by another rule called the $*$-*property* (pronounced "star property") that says that a subject may have write access to an

[1]Matters are complicated in practice by the use of compartments (e.g., NATO, NOFORN) in combination with the basic classifications to create a partial ordering.

object only if the object's label is equal to or greater than that of the subject. The combination of the simple and the * properties (i.e., a subject can only read "down" and write "up" in security level) constitute the historically significant Bell and La Padula security policy [10]. Under further examination, this policy raises important questions that will be considered shortly. First, though, we return to the related question of completeness of an access control policy.

The access control policy of the Unix file system can be bypassed if users can directly read or write the contents of the disk on which those files are stored. Thus, although our interest is in protecting files, we also need to be concerned about the disk, and possibly other elements of the system as well. So the issue of completeness in access control concerns how much of the system needs to be placed under access control, and in what way, for us to be sure that the resource we actually want to protect is, indeed, protected against all possible attacks. This issue is complicated by the fact that security is really about protecting *information*, not mere data, so that any channel (a metaphorical example would be by tapping on the walls in Morse code) that allows the information content of a file to be conveyed to an unauthorized user is as dangerous as the ability to copy a file directly.

The possible channels for information flow can be quite subtle and hard to detect (there were at least two in Bell and La Padula's "Multics Interpretation" [10]). For example, suppose we had a special Unix system that imposed the Bell and La Padula policy on file access, but with the additional property that file names are required to be unique across all users: an attempt to create a file with an existing name returns an error code. Then, a SECRET process can convey information to an UNCLASSIFIED one by creating files with prearranged names: the UNCLASSIFIED process retrieves the information by checking whether or not it is able to create files with those names. This is an example of a "covert" channel; more particularly, it is a covert *storage* channel (because it exploits information stored in the directory structure of the file system; the other kind of channel uses *timing* information—see Section 4.3) [60, 65]. The channel is noisy (some other, innocent, process might have created files with those names), but coding techniques allow information to be transmitted reliably over noisy channels. Covert channels are of concern for two reasons: first, they can be used to transmit information at surprisingly high bandwidth (one early demonstration drove a teletype at full speed using a channel that depended on sensing where a disk head was positioned [95]) and second, they are no different in concept from more blatant channels (e.g., the unprotected disk) that leave a resource open to direct access (both are symptoms of incompleteness)—so that unless we have methods of specification and verification that are able to eliminate subtle covert channels, we have little guarantee that we can eliminate any channels at all.

It might seem that information flow and covert channels are esoteric security concerns and that only basic access control is relevant to partitioning. However,

while it is true that covert information flow may be of little concern for partitioning (because it depends on collusion between sender and receiver and is therefore implausible as a fault manifestation), the *mechanisms* used for such flow definitely are of concern. Consider, for example, the unique-file-name channel described above. This serves as a channel for information flow because one subject can affect the behavior perceived by another (i.e., whether or not the attempt to create a file returns an error), and this is surely contrary to the expectations of partitioning—for one interpretation of those expectations is that the behavior perceived by software in any given partition should be independent of the actions by software in other partitions. We might try to arrange for this expectation to be satisfied in the presence of the unique-file-name restriction by allocating disjoint name spaces to each partition. But then a fault in the software of one partition could cause it to create a file from another's name space—and thereby cause a subsequent file creation in that other partition to fail. This example shows that covert channels for information flow raise issues that are relevant to partitioning, and that examination of how security has dealt with these channels may be of use in partitioning.

Another potential problem with access control formulations of security is that they depend on informal understandings of what "read" and "write" accesses really mean. We can construct perverse systems in which these terms are given incorrect (e.g., reversed) interpretations and that satisfy the letter of an access control policy while violating its spirit [72].

Covert channels and perverse interpretations are both symptoms of the real problem with access control as we have used it: it is a mechanism for *implementing*, not an instrument for *specifying*, security policies. An adequate specification should get at the "intent" that underlies a security policy in a convincing manner. It should then be possible to prove that an implementation in terms of access control correctly enforces the policy. Problems of completeness, covert channels, and perverse interpretations should all be eliminated by a sound approach of this kind. The next section examines such approaches.

4.1.2 Noninterference

To repair the problems with access control, we need to be more explicit about our system model: we need to specify how a system computes and interacts with its environment, how inputs and outputs are observed, and how subjects and objects are identified. Then we can specify security in terms of constraints on the observable behavior of the system, without needing to describe mechanisms to enforce those constraints (although we would hope to be able to describe such mechanisms at a later stage of development, and to verify that they enforce the desired policy).

The most successful treatments of this kind are all variations on a formulation called *noninterference* that was introduced by Goguen and Meseguer in 1992 [34],

although the key idea was adumbrated five years earlier [31]. That key idea is that if there is no flow of information from one security classification to another, then the behavior perceived by subjects of the second ("lower") classification should be independent of any activity by subjects of the first ("higher") classification. In particular, the behavior perceived by the second classification should be unchanged if all activity by the first is removed. The precise details depend on the formal model of computation employed, but the traditional treatment uses a finite automaton as the system model: the automaton changes state and produces an output in response to inputs, which are labeled with their security level. A relation $p \rightsquigarrow q$ indicates whether level p is allowed to convey information to or *interfere* with level q; its negation is the *noninterference* relation $\not\rightsquigarrow$, which is considered a specification of the desired *security policy*. A sequence of inputs α is *purged* for level p by removing all inputs from levels that may not interfere with p; this purged input sequence is denoted α/p. Starting from some initial state s_0, the state of the automaton after consuming the input sequence α is $run(s_0, \alpha)$, while that after consuming the purged sequence is $run(s_0, \alpha/p)$. The noninterference formulation of security then requires that any level p input must produce the same output in both these states. The intuition is that this ensures that no experiment conducted at level p can reveal anything about the presence or absence of earlier inputs from levels that should not interfere with p.

The noninterference formulation of security is stated in terms of a system's behavior in response to a *sequence* of inputs. An *unwinding theorem* reduces this to three conditions on its behavior with respect to *individual* inputs. These conditions are stated in terms of each level's "view" of the system state (intuitively, if the system state is thought of as consisting of different components "belonging" to each level, then level p's view of the state comprises its own component and the components of all the levels that are allowed to interfere with p). If the level p views of two states are the same, we say these states "look the same to p" (technically, this is an equivalence relation on states).

Output Consistency: if two states look the same to p, then a level p input must produce the same output in both states.

Step Consistency: if two states look the same to p, then the states that result from applying the same input (of any level) to both states must also look the same to p.

Local Respect (for $\rightsquigarrow$): the system state must look the same to p before and after an input of a level that is noninterfering with p.

It is straightforward to prove that these conditions are sufficient to imply noninterference. The proof is formalized and mechanically verified in one of the tutorials for the PVS verification system [94].

A connection between the noninterference and access control notions of security can be established by interpreting the unwinding conditions in access control terms. We suppose that the system state is a function from *objects* to *values* and that each object has a level. Inputs of level p are reinterpreted as *actions* performed by a *subject* of level p. Then we suppose that access control enforces the following *Reference Monitor Assumptions.*

- The output produced by an action depends only on the values of objects to which the subject performing the action has read access.
- If an action changes the value of any object, then its new value depends only on the values of objects to which the performing subject has read access.
- An action may change the values only of objects to which the performing subject has write access.

With these assumptions, access control can enforce the unwinding conditions by setting up the controls as follows (these are essentially the Bell and La Padula conditions).

1. If $p \rightsquigarrow q$, then the objects to which subjects of level p have read access must be a subset of those to which subjects of q have read access, and
2. A subject of level p may have write access to an object for which a subject of level q has read access only if $p \rightsquigarrow q$.

The connection between the two formulations is established by interpreting a subject's "view" as the values of all the objects to which it has read access. A proof is given in [90, Section 2.1]. The proof requires formalizing the reference monitor assumptions, which is surprisingly difficult to do correctly (Popek and Farber [83], who first recognized the importance of these conditions, made errors in formalizing them).

Contrary to early expectations (e.g., [35]), standard noninterference requires the interferes relation $\rightsquigarrow$ to be transitive [90]. All such transitive relations are equivalent to multilevel security policies, and the two conditions on access control enumerated in the previous paragraph are likewise equivalent to the Bell and La Padula properties in these cases [90, Section 3.1].

Because they imply a partial ordering on security levels, multilevel security policies do not seem to capture the concerns of partitioning all that closely, but *intransitive* policies (that is, those where $\rightsquigarrow$ is not required to be transitive) seem more promising. Intransitive policies capture the additional security restrictions known as *channel control* [88] or *type enforcement* [13], which are concerned not only with whether information may flow from one place to another, but with the paths through which it may flow. Channel control security policies can be represented by directed

graphs, where nodes represent security domains and edges indicate the direct information flows that are allowed. The paradigmatic example of a channel-control problem is a controller for end-to-end encryption, as portrayed in Figure 4.1.

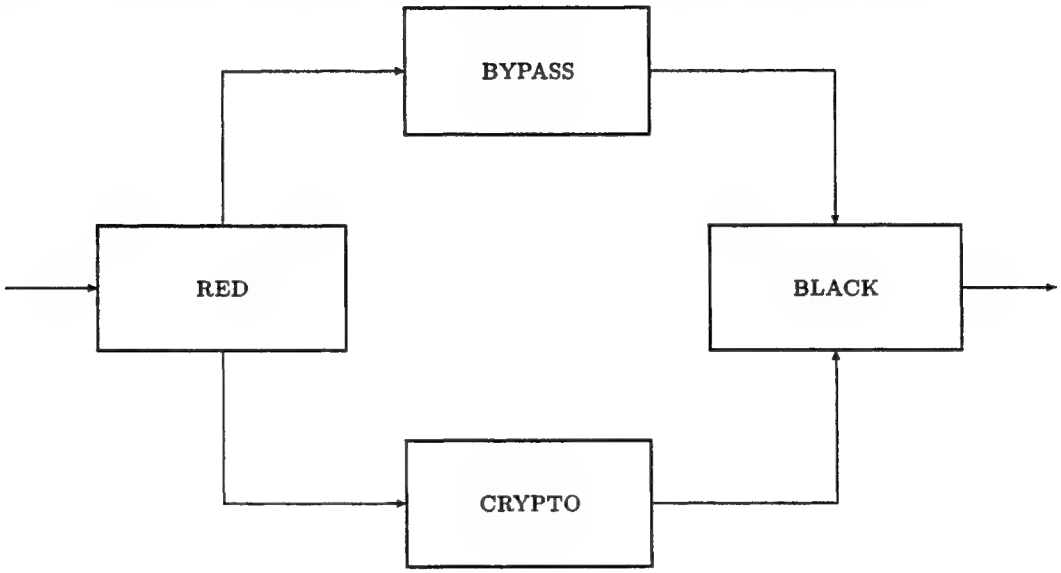


Figure 4.1: Allowed Information Flows for an Encryption Controller

Plaintext messages arrive at the RED side of the controller; their bodies are sent through the encryption device (CRYPTO); their headers, which must remain in plaintext so that network switches can interpret them, are sent through the BYPASS. Headers and encrypted bodies are reassembled in the BLACK side and sent out onto the network. The security policy we would like to specify here is the requirement that the *only* channels for information flow from RED to BLACK must be those through the CRYPTO and the BYPASS (it is a separate problem to specify what those components must do). Notice that the edges indicating allowed information flows in this example are not transitive: information is allowed to flow from RED to BLACK via the CRYPTO and BYPASS, but must not do so directly.

Noninterference can be extended to intransitive policies by substituting a more complicated *purge* function for the standard one. When $p \not\rightsquigarrow q$, the usual requirement is that deleting all actions performed by p should produce no change in the behavior of the system as perceived by q. This is too strong if we also have the assertions $p \rightsquigarrow r$ and $r \rightsquigarrow q$. Surely we should only delete those actions of p that are not followed by actions of r (in the CRYPTO example, RED, BLACK, and BYPASS or CRYPTO take the roles of p, q, r, respectively). This insight, and a definition of the generalized *purge* function, were given by Haigh and Young [38], together with corre-

sponding unwinding conditions. Unfortunately, one of their unwinding conditions is incorrect; correct conditions were given, and formally verified, by Rushby [90]. These unwinding conditions simply replace the step consistency condition by a weak form.

Weak Step Consistency: if two states look the same to p, and also look the same to q, then the states that result from applying the same input of level q to both states must also look the same to p.

The corresponding conditions for access control enforcement consist simply of the second of the two conditions given on page 66.[2]

The formal statements of standard and intransitive noninterference use an automaton as their formal system model and therefore apply straightforwardly only to a single monolithic system. To examine the interactions of multiple, distributed systems, more general models are required—for example, transition relations or process algebras—and it is necessary to admit nondeterminism. Nondeterminism arises naturally in concurrent systems, because there is generally no system-wide coordination of the rate at which different components proceed; hence, interactions can occur in different orders in otherwise identical runs, and the behaviors perceived in those runs can diverge markedly (this is why it is so hard to debug concurrent systems). To accommodate this system-level nondeterminism, noninterference for concurrent systems is formulated to require that the *set* of behaviors possible in a given scenario is unchanged at a given level when interactions are purged in some suitable way. One problem with this formulation is that it does not define a *property* in the technical sense.

A system can be identified with the set of runs that it can produce (a run is a sequence or "trace" of inputs, outputs, and other significant interactions). A specification is likewise a set of runs, and a system satisfies a specification if its runs are a subset of those of the specification. A set of runs is called a *property*, so that specifications and systems can both be considered properties. Special classes of properties called *safety* and *liveness* play an important role in formal methods of analysis, and it can be shown that every property can be expressed as the conjunction of a safety and a liveness property [5]. Security, however, is not a property in this sense: it is not a set of runs, but a set of sets of runs [73]. This means that standard methods for deriving or verifying an implementation that satisfies a given specification do not work for security—because these methods apply only to properties.

Another problem when noninterference is extended to concurrent systems in the manner just described is that it is not *compositional*: that is, two systems individually satisfying some noninterference policy can be combined to yield a composite

[2]This might seem to suggest that the first condition on page 66 is implied by the second when the policy is transitive. In fact, this is not necessarily so for a given set of access controls, but it will be possible to construct another set (i.e., a different assignment of read and write permissions) that will satisfy both conditions. This is a consequence of the "nesting property" for transitive policies [90, Theorem 5].

system that does not satisfy that policy [71]. Many alternative formulations of noninterference were proposed for concurrent systems in the attempt to overcome this unattractive result. Unfortunately, those that were compositional were either very unintuitive (having no plausible interpretation as a natural security concern), or were excessively restrictive (and unlikely to be satisfied by practical systems). A partial resolution was provided by Roscoe, who suggested that the difficulty was due to a failure to appreciate the significance of nondeterminism when contemplating security [86].

The problem with nondeterminism is that it can sometimes be resolved in a way that depends on unsecure information flow. A typical example would be a system with two levels, LOW and HIGH where HIGH is required to be noninterfering with LOW. Inputs to LOW cause the outputs *odd* or *even* to be generated nondeterministically *unless* there have been any high inputs, in which case the LOW output is *odd* or *even* according to the oddness or evenness of the last HIGH input (the HIGH inputs are assumed to be positive integers). This example satisfies most definitions of noninterference for concurrent systems because the set of *possible* behaviors observable at the LOW level is unchanged by the presence or absence of HIGH-level activity—yet it plainly violates any reasonable interpretation of "secure system." The violation is exposed when the system is composed with one that generates only even numbers on the HIGH input. Roscoe excluded such paradoxical constructions by requiring their component systems to have behavior that is deterministic at each security level. Roscoe's insistence on determinism also suggests a resolution to another difficulty that had plagued most earlier treatments: noninterference is not preserved under refinement. Refinement in this (process algebra) context means a reduction in nondeterminism, and it poses the same challenge to noninterference as composition. Roscoe's treatment is couched in the formalism of CSP [40], where a process is deterministic if it is free of "divergence" and never has a choice between "accepting" and "refusing" an event [87]. The relationship between this treatment and traditional interpretations of determinism and security in state machines is one that requires clarification.

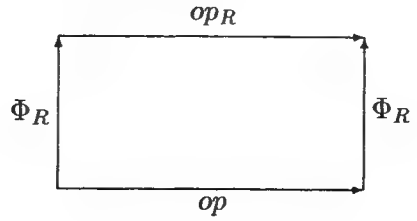
There is another sense of refinement for which security in general (not only its noninterference formulations) is not preserved. This is the notion of refinement in the sense of elaboration, where more mechanisms and details are added to a specification in order to obtain an implementation that is effectively executable. Under the standard notion of correctness for such refinements, it is necessary only to show that the properties of the specification are implied by those of the implementation: the implementation is required to do *at least* as much as the specification, but it is not prohibited from doing *more*. An implementation of the specification suggested by Figure 4.1, for example, must provide at least the four communications channels shown, but the standard notion of correct refinement would not prevent it adding a direct communications channel between RED and BLACK—despite the fact that the

absence of such a channel is the whole point of the design. For security, it is necessary to constrain the notion of correct refinement so that the implementation does not add capabilities that are absent in the specification. Clearly the implementation must contain more details and mechanisms than the specification (else it is surely not an implementation), but for secure refinement these mechanisms and details must have no consequences on the behavior that can be perceived at the originally specified interfaces. The formal characterization of this requirement is given in terms of *faithful interpretations* and is due to Moriconi, Qian, Riemenschneider, and Gong [75].

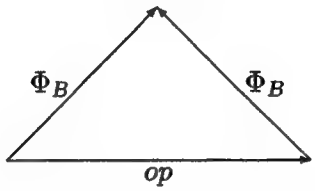
4.1.3 Separability

Using Roscoe's perspective, an adequate treatment for distributed channel-control security might be achieved by taking the nondeterministic composition of deterministic systems, each characterized by intransitive noninterference. Some architectural refinement to a more detailed implementation level could be obtained using faithful interpretations, and the restrictions within each system could then be enforced by access control, using the derivation from the unwinding conditions described earlier. (As far as I know, nobody has determined whether the formal details of the various models support this combination, nor whether satisfactory properties can be derived for the combination, but it seems plausible.) However, the resulting model would still be rather abstract for the purpose of deriving, for example, conditions on how a single processor should implement the RED, BYPASS, and BLACK components of Figure 4.1 (the CRYPTO is usually an external device).

An approach called *separability* was proposed for this problem by Rushby [88]. The idea is easiest to understand when no communications are allowed between the separate components. Then the idea is that the implementation should provide the appearance of a separate, dedicated processor to each component. The real processor is time shared, so that it sometimes performs instructions on behalf of one component and sometimes on behalf of another. The requirements for separability can be expressed in terms of *abstraction functions* that give the "view" of the processor perceived by each component. For example, if we have just two components, RED and BLACK, and Φ_R and Φ_B denote their respective abstraction functions, then the requirement when the processor is executing instructions on behalf of RED is that the following diagram should commute.



That is to say, the state change in the physical processor caused by executing the instruction *op* should be consistent with execution of the "abstract" operation op_R on RED's view of the processor. At the same time, BLACK's view of the processor should be unchanged, as expressed by the following diagram.



Because I/O devices can directly observe and change aspects of the real processor's internal state (by reading and writing its device registers, for example), and can also influence its instruction sequencing mechanism (by raising interrupts), the activity of these devices is relevant to security. Consequently, we must impose conditions on their behavior. Expressed informally (and only from the RED component's point of view), these conditions are the following.

1. If $\Phi_R(\sigma) = \Phi_R(\tau)$ and activity by a RED I/O device changes the state of the real processor from σ to σ', and the same activity would also change it from τ to τ', then $\Phi_R(\sigma') = \Phi_R(\tau')$ (i.e., state changes in the RED view caused by RED I/O activity must depend only on the activity itself and the previous state of the RED view).

2. If activity by a non-RED I/O device changes the state of the real processor from σ to τ, then $\Phi_R(\sigma) = \Phi_R(\tau)$ (i.e., non-RED I/O devices cannot change the state of the RED view).

3. If $\Phi_R(\sigma) = \Phi_R(\tau)$, then any outputs produced by RED I/O devices must be the same in both cases.

4. If $\Phi_R(\sigma) = \Phi_R(\tau)$, then the next operation executed on behalf of the RED component must also be the same in both cases.

Separability was proposed before formal treatments of concurrent systems had been fully developed, so the justification of the above conditions presented in [89] is not fully satisfactory. Furthermore, neither the informal nor the formal presentation deals with allowed communications channels between components. The proposal in [88] is to remove the mechanisms intended to provide the desired communications channels and then verify, using the conditions above, that the components of the resulting system are isolated. Jacob [48] noted that this does not exclude a particular kind of covert channel (called a "legitimate" channel) that piggybacks undesired clandestine communication on the desired channel.

A more modern treatment [93] derives the conditions for separability with communications from those for intransitive noninterference. This treatment weakens the "triangular" commutative diagram of strict separability so that it applies only if RED $\not\rightsquigarrow$ BLACK (this derives from the "local respect for $\not\rightsquigarrow$" unwinding condition) and, when RED $\rightsquigarrow$ BLACK, it replaces the "rectangular" diagram by the following condition (which is based on the "weak step consistency" unwinding condition).

$$\Phi_R(\sigma) = \Phi_R(\tau) \wedge \Phi_B(\sigma) = \Phi_B(\tau) \supset \Phi_B(op(\sigma)) = \Phi_B(op(\tau))$$

Notice that this last condition does not use the abstract operation op_R that appears in the "rectangular" commutative diagram. This is because we do not really care what this operation is, only that $\Phi_B(op(\sigma))$ should be functional in $\Phi_B(\sigma)$, and the formula above expresses this directly.

4.2 Integrity Policies

The previous sections have considered computer security notions related to the undesired *disclosure* of information. There are similar notions related to the *modification* of information, where the main concern is to ensure *integrity* of the protected information. Integrity is related to the "reliability" or "quality" of information: information of high integrity should not be allowed to become contaminated by information of low integrity. This requirement can be treated as a strict dual to the Bell and La Padula security policy (that is, a subject can only read "up" and write "down" in integrity level), and is known as the Biba integrity policy [12].

Clark and Wilson [15] argued that the integrity of information is also a function of the operations that are performed on it, and the identity of those who invoke those operations. A user should not be able to invoke arbitrary operations on high-integrity information, but only certain *well-formed transactions*, and the admissible transactions might be determined by the state of the data, the identity of the user, and other factors. In commercial environments, the transactions available to a user are often governed by requirements for *separation of duties*: a user who authorizes a purchase should not be the same as the one who selects the vendor.

Other similar models for integrity have been proposed, and there has been considerable investigation of whether these or the Clark-Wilson model can be enforced by adaptations of security mechanisms developed to control disclosure [79].

4.3 Timing Channels and Denial of Service

Most work on formalizing security has focused on the data and information flow issues described in the previous sections. In partitioning terms, these all concern issues in spatial partitioning. There are, however, two topics in computer security

that correspond to issues in temporal partitioning: timing channels and denial of service.

Timing channels (they were called "covert channels" when first identified [60]) are mechanisms for clandestine information flow that work by modulating the time when some events occur, or the rate at which they occur. For example, a process can choose whether or not to give up its time slice early. If only two processes are running, the other process can use the time at which it receives control to infer the choice made by the other process [60]. More generally, the decisions of a real-time scheduler can be manipulated to provide a channel for information flow [16]. Other timing channels modulate the load or contention on some system resource (e.g., the system bus [43]) or parameters affecting performance (e.g., the time to seek a disk track is affected by whether the previous seek was to a nearby or distant track [50]; the time to access a memory page will be affected by whether or not it was previously swapped out to disk [95]).

Where they cannot be removed, timing channels are typically rendered harmless either by reducing contention, or by introducing randomness into the behavior of the resource being manipulated [37, 68, 104], or by reducing the precision of the various "clocks" (e.g., time-of-day clocks, timers, instruction loops, asynchronous I/O performance) by which a process can measure the passage of time [43]. These measures do not block a timing channel, but they introduce sufficient noise that its bandwidth is reduced to acceptable levels (typically less than 10 bits per second).

Whereas the concerns of partitioning and security are quite close in the case of storage channels, they diverge for timing channels. The very existence of a timing channel is unacceptable in a partitioned system, since it indicates that one partition can change the temporal behavior observed by another. Similarly, the remedies used in security to reduce the bandwidth of timing channels are worse than the original problem from the perspective of partitioning, because they introduce further unpredictability into system behavior.

Formal analysis of pure timing channels is generally based on information theory (e.g., [76, 77]), but there is dispute over whether some channels (e.g., the disk arm channel) really are timing channels, storage channels, or a combination of the two [112]. Consequently, formal description and analysis of such channels is difficult, and informal methods are generally employed. As described in Section 3.1.2, static partition scheduling requires implementation choices (strict determinism, no concurrent I/O) that eliminate the mechanisms that could serve as timing channels. In systems that do not require such strict temporal partitioning, the techniques used in computer security to identify timing channels [112] might help reveal unexpected sources of temporal interference.

Denial of service can be seen as an extreme type of timing channel: the perceived performance of some resource is reduced to an unacceptable level, rather than merely modulated. In the limit, the resource may become unavailable to some processes.

The possibility of this limiting case is usually equivalent to the existence of a storage channel. For example, if file space is shared between two processes, then one can deny service to the other by consuming all available space—but this is also a channel by which one process can convey information to another (the receiving channel attempts to create a file: success is taken as a 1 bit, failure as a 0; the transmitting process determines the outcome by consuming and releasing space). Because denial of service is related to timing and storage channels, it can be prevented by enforcing strict spatial and temporal partitioning. In general-purpose systems, the strictness of these mechanisms may be considered undesirable: they would require, for example, fixed per-process allocations of file space. Attempts to provide flexible resource allocation without incurring the risk of denial of service require "user agreements" that place limits on the demands that each process may place on each resource and that are enforced by a "resource allocation monitor" or "denial of service protection base" [64, 74] (these are somewhat similar to the quality of service ideas used in multimedia systems [102]). Formalizations of these approaches are stated in terms of fair or maximum waiting times [33, 113].

These more elaborate treatments of denial of service are probably unacceptable in strictly partitioned systems because they still allow the response perceived by one application to be influenced, even if not denied, by another. They may also be unnecessary in partitioned systems because the requirements for temporal partitioning seem stronger than those for denial of service: thus, denial of service should automatically be excluded in any system that provides strict temporal partitioning. Formal justification for this claim would be an interesting and worthwhile exercise.

4.4 Application to Partitioning

The formal models for computer security reviewed in the previous sections provide several ideas that seem applicable to partitioning. In particular, the central idea of noninterference—that the behavior perceived at one security level should be independent of actions at higher levels—can be reinterpreted in the context of partitioning and fault tolerance by supposing that ordinary behavior should be independent of faults: that is, faults are actions invoked by the environment, which is at a level that should be noninterfering with the level of ordinary users. This approach has been explored by Weber and by Simpson [99, 100, 108, 109]. It works well as a specification for partitioning when the partitions are completely isolated (in which case it is equivalent to the strict form of separability): if we have two partitions A and B that do not communicate in any way, then saying that the behavior of B must be independent of that of A is a good way to say that faults in A must not affect B. It works less well when A has to communicate with B: noninterference says only that A interferes with B and does not discriminate between legitimate interference

(the known communication stream) and illegitimate (e.g., changes to B's private data).

This example shows that the concerns of security are, in a certain sense, too coarse to capture those of partitioning: security is concerned only with *whether* information can flow from A to B, not with *how* the flow can affect B. Channel control and its formalization by intransitive noninterference does allow the desired discrimination, but only at the cost of introducing a third component C to represent the buffer used for the intended A to B communication stream. Using intransitive flows, we would specify $A \rightsquigarrow C \rightsquigarrow B$ and $A \not\rightsquigarrow B$. This approach seems to capture some of the concerns of partitioning, but the introduction of the third component is artificial and unattractive.

A more fundamental objection to the idea that noninterference can serve as a model for partitioning is that partitioning is a safety property (because violations of partitioning occur at specific points in specific runs) whereas noninterference is not even a "property" (recall page 68). This suggests that noninterference is an unnecessarily subtle notion for partitioning, and that something simpler should suffice.

There is another sense in which the concerns of security diverge from those of partitioning: security assumes that all components are untrustworthy and that the mechanisms of security must be set up so that only allowed information flows occur, no matter how the components behave. In partitioning, however, we are concerned only with misbehavior by faulty partitions and are willing to trust nonfaulty components to safeguard their own interests. For example, suppose that two components A and B are statically scheduled and that each begins execution at a known entry point each time it is scheduled (this is the restart model of partition swapping). Suppose further that each has an area of "scratchpad" memory that is assumed to be "dirty" at the start of each execution: that is, the software in each of A and B is verified to perform its functions with no assumptions on the initial contents of the scratchpad memory. Finally, suppose that A and B are required to be isolated from one another. Then the scratchpad can be shared between A and B under the partitioning interpretation of isolation, but not under the corresponding security interpretation. The reason is that when B receives control, the scratchpad may contain data written by A; under the security interpretation we may assume nothing about even a nonfaulty B (in particular, that it will not "peek" at the data left by A), and so the scratchpad is a channel for information flow from A to B in violation of the isolation security policy. In the partitioned system, we accept (or specify) that a nonfaulty B does not do this, and our concern is to be sure that A (even if faulty) does not write outside its own memory or the scratchpad. Notice that this arrangement would not be safe in the restoration model of partition swapping, because A could preempt B, change its scratchpad, and then allow B to resume.

These examples demonstrate that the concerns of partitioning and security, although related, do not coincide. Thus, although formal treatments of partitioning may possibly be developed using ideas from computer security, they cannot be based directly on existing security models. Research to develop formal models of partitioning, and to refine the distinctions between partitioning and security, would be illuminating for both fields.

Chapter 5

Conclusion

We have reviewed some of the motivation for integrated modular avionics and the requirement for partitioning in such architectures. We then considered mechanisms for achieving partitioning, the interactions between these mechanisms and those for system structuring, scheduling, and fault tolerance, and issues in providing assurance for partitioning. Finally, we reviewed work in computer security that has similar motivation to partitioning.

Although partitioning is a very strong requirement and imposes many restrictions, there is a surprisingly wide range of architectural choices that can achieve adequate partitioning. The space of these design choices is seen most clearly in scheduling, where both static and dynamic schedules seem able to combine flexibility with highly assured partitioning.

The strongest need for future work is to develop the narrative description given here into a mathematical framework that will permit rigorous analysis of architectural choices for partitioned systems and provide a strong basis for the assurance of individual designs. There is already some significant work in this direction [24, 25, 27, 111], but great opportunities remain, particularly with respect to distributed systems and temporal partitioning. We are examining these topics in current work and will describe our results in a successor to this report.

References

[1] *ARINC Specification 651: Design Guidance for Integrated Modular Avionics.* Aeronautical Radio, Inc., Annapolis, MD, November 1991. Prepared by the Airlines Electronic Engineering Committee.

[2] *ARINC Specification 659: Backplane Data Bus.* Aeronautical Radio, Inc., Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.

[3] *ARINC Specification 629: Multi-Transmitter Data Bus; Part 1, Technical Description (with five supplements); Part 2, Application Guide (with one supplement).* Aeronautical Radio, Inc., Annapolis, MD, December 1995/6. Prepared by the Airlines Electronic Engineering Committee.

[4] *ARINC Specification 653: Avionics Application Software Standard Interface.* Aeronautical Radio, Inc., Annapolis, MD, January 1997. Prepared by the Airlines Electronic Engineering Committee.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[6] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, January 1980.

[7] C. R. Attanasio, P. W. Markstein, and R. J. Phillips. Penetrating an operating system: A study of VM/370 integrity. *IBM Systems Journal*, 15(1):102–116, 1976.

[8] Algirdas Avižienis and Yutao He. Microprocessor entomology: A taxonomy of design faults in COTS microprocessors. In Weinstock and Rushby [110], pages 3–23.

[9] Henry M. Ballard, David M. Bicksler, Thomas Taylor, and H. O. Lubbes. Ensuring process security in the ALS/N environment. In *COMPASS '86*

(Proceedings of the First Annual Conference on Computer Assurance), pages 60–68, IEEE Washington Section, Washington, DC, July 1986.

[10] D. E. Bell and L. J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, Bedford, MA, March 1976.

[11] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO, December 1995. (ACM Operating Systems Review, Vol. 29, No. 5).

[12] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR 3153, Mitre Corporation, Bedford, MA, June 1975.

[13] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th DoD/NBS Computer Security Initiative Conference*, pages 18–27, Gaithersburg, MD, September 1985.

[14] Mikhail Chernyshov. Post-mortem on failure. *Nature*, 339:9, May 4, 1989.

[15] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the Symposium on Security and Privacy*, pages 184–194, IEEE Computer Society, Oakland, CA, April 1987.

[16] Raymond K. Clark, Douglas M. Wells, Ira B. Greenberg, Peter K. Boucher, Teresa F. Lunt, Peter G. Neumann, and E. Douglas Jensen. Effects of multilevel security on real-time applications. In *Proceedings of the Ninth Annual Computer Security Applications Conference*, pages 120–129, IEEE Computer Society, Orlando, FL, December 1993.

[17] Henry S. F. Cooper Jr. Annals of space (the planetary community)—part 1: Phobos. *New Yorker*, pages 50–84, June 11, 1990.

[18] Henry S. F. Cooper Jr. *The Evening Star: Venus Observed.* Farrar Straus Giroux, New York, NY, 1993.

[19] Robert D. Culp and George Bickley, editors. *Proceedings of the Annual Rocky Mountain Guidance and Control Conference, Advances in the Astronautical Sciences*, Keystone, CO, February 1993. American Astronautical Society.

[20] Sadegh Davari and Lui Sha. Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, 26(2):110–120, April 1992.

[21] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[22] David L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, Association for Computing Machinery, San Diego, CA, January 1996.

[23] *Profibus Standard: DIN 19245.* Deutsche Industrie Norm, Berlin, Germany, 1995. Two volumes; see also `http://www.profibus.com`.

[24] Ben L. Di Vito. A formal model of partitioning for integrated modular avionics. NASA Contractor Report CR-1998-208703, NASA Langley Research Center, August 1998.

[25] Ben L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In Weinstock and Rushby [110], pages 269–286.

[26] Eileen M. Dukes. Magellan attitude control mission operations. In Culp and Bickley [19], pages 375–388.

[27] Bruno Dutertre and Victoria Stavridou. A model of non-interference for integrating mixed-criticality software components. In Weinstock and Rushby [110], pages 301–316.

[28] The interfaces between flightcrews and modern flight deck systems. Report of the FAA human factors team, Federal Aviation Administration, 1995. Available at `http://www.faa.gov/avr/afs/interfac.pdf`.

[29] *System Design and Analysis.* Federal Aviation Administration, June 21, 1988. Advisory Circular 25.1309-1A.

[30] *RTCA Inc., Document RTCA/DO-178B.* Federal Aviation Administration, January 11, 1993. Advisory Circular 20-115B.

[31] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *Sixth ACM Symposium on Operating System Principles*, pages 57–65, November 1977.

[32] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In SOSP-16 [101], pages 38–51.

[33] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering*, SE-10(3):320–324, May 1984.

[34] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20, IEEE Computer Society, Oakland, CA, April 1982.

[35] J. A. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the Symposium on Security and Privacy*, pages 75–86, IEEE Computer Society, Oakland, CA, April 1984.

[36] B. D. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proceedings of the Symposium on Security and Privacy*, pages 13–23, IEEE Computer Society, Oakland, CA, April 1984.

[37] James W. Gray, III. On introducing noise into the bus-contention channel. In SSP'93 [46], pages 90–98.

[38] J. Thomas Haigh and William D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, SE-13(2):141–150, February 1987.

[39] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of μ-kernel-based systems. In SOSP-16 [101], pages 66–77.

[40] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[41] Harry Hopkins. Fit and forget fly-by-wire. *Flight International*, pages 89–92, December 3, 1988.

[42] Kenneth Hoyme and Kevin Driscoll. SAFEbus™. *IEEE Aerospace and Electronic Systems Magazine*, 8(3):34–39, March 1993.

[43] Wei-Ming Hu. Reducing timing channels with fuzzy time. In SSP'91 [45], pages 8–20.

[44] M. Huguet. The protection of the processor status word of the PDP-11/60. *ACM Computer Architecture News*, 10(4):27–30, June 1982.

[45] *Proceedings of the Symposium on Security and Privacy*, Oakland, CA, May 1991. IEEE Computer Society.

[46] *Proceedings of the Symposium on Security and Privacy*, Oakland, CA, May 1993. IEEE Computer Society.

[47] *ISO Standard 11898: Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication.* International Standards Organization, Switzerland, November 1993.

[48] Jeremy Jacob. A note on the use of separability for the detection of covert channels. *Cipher (Newsletter of the IEEE Technical Committee on Security and Privacy)*, pages 25–33, Summer 1989.

[49] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In SOSP-16 [101], pages 52–65.

[50] Paul A. Karger and John C. Wray. Storage channels in disk arm optimization. In SSP'91 [45], pages 52–61.

[51] Rick Kasuda and Donna Sexton Packard. Spacecraft fault tolerance: The Magellan experience. In Culp and Bickley [19], pages 249–267.

[52] Philip J. Koopman, Jr. Perils of the PC cache. *Embedded Systems Programming*, 6(5):26–34, May 1993.

[53] Herman Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. In *6th IEEE Workshop on Future Trends in Distributed Computing*, pages 310–315, IEEE Computer Society, Tunis, Tunisia, October 1997.

[54] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered? *IEICE Transactions on Information and Systems*, E76-D(11):1325–1332, November 1993. Institute of Electronics, Information, and Communications Engineers, Japan.

[55] Hermann Kopetz. *Real-Time Systems: Design Princples for Distributed Embedded Applications.* The Kluwer International Series in Engineering and Computer Science. Kluwer, Dordrecht, The Netherlands, 1997.

[56] Hermann Kopetz. A comparison of CAN and TTP. Technical report, Technische Universität Wien, Vienna, Austria, March 1998.

[57] Hermann Kopetz. The time-triggered (TT) model of computation. Technical report, Technische Universität Wien, Vienna, Austria, March 1998. Published in RTSS'98.

[58] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[59] A. A. Lambregts. Automatic flight controls: Concepts and methods. Draft paper by FAA National Resource Specialist for Advanced Controls, January 1998.

[60] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[61] C. E. Landwehr. A survey of formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[62] John P. Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm—exact characterization and average case behavior. In *Real Time Systems Symposium*, pages 166–171, IEEE Computer Society, Santa Monica, CA, December 1989.

[63] K. Rustan M. Leino and Rajeev Joshi. A semantic approach to secure information flow. Technical Report 032, Digital Systems Research Center, Palo Alto, CA, December 1997.

[64] Jussipekka Leiwo and Yuliang Zheng. A method to implement a denial of service protection base. In Vijay Varadharajan, Josef Pieprzyk, and Yi Mu, editors, *Information Security and Privacy: Second Australasian Conference (ACISP '97)*, Volume 1270 of Springer-Verlag *Lecture Notes in Computer Science*, pages 90–101, Sydney, Australia, July 1997.

[65] S. B. Lipner. A comment on the confinement problem. In *Fifth ACM Symposium on Operating System Principles.* pages 192–196, ACM, 1975.

[66] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[67] C. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[68] Keith Loepere. Resolving covert channels within a B2 class secure system. *ACM Operating Systems Review*, 19(3):9–28, July 1985.

[69] Yoshifumi Manabe and Shigemi Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2):171–181, March 1998.

[70] R. A. Mayer and L. H. Seawright. A virtual machine time sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[71] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the Symposium on Security and Privacy*, pages 161–166, IEEE Computer Society, Oakland, CA, April 1987.

[72] John McLean. A comment on the "basic security theorem" of Bell and La Padula. *Information Processing Letters*, 20:67–70, 1985.

[73] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 79–93, IEEE Computer Society, Oakland, CA, May 1994.

[74] Jonathan K. Millen. A resource allocation model for denial of service. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 137–147, IEEE Computer Society, Oakland, CA, May 1992.

[75] Mark Moriconi, Xiaolei Qian, R. A. Riemenschneider, and Li Gong. Secure software architectures. In *Proceedings of the Symposium on Security and Privacy*, pages 84–93, IEEE Computer Society, Oakland, CA, May 1997.

[76] Ira S. Moskowitz, Steven J. Greenwald, and Myong H. Kang. An analysis of the timed Z-channel. In *Proceedings of the Symposium on Security and Privacy*, pages 2–31, IEEE Computer Society, Oakland, CA, May 1996.

[77] Ira S. Moskowitz and Alan R. Miller. Simple timing channels. In *Proceedings of the Symposium on Security and Privacy*, pages 56–64, IEEE Computer Society, Oakland, CA, May 1994.

[78] A Nadesakumar, R. M. Crowder, and C. J. Harris. Advanced system concepts or future civil aircraft—an overview of avionic architectures. *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, 209:265–272, 1995.

[79] *Integrity in Automated Information Systems.* National Computer Security Center, September 1991. Technical Report 79-91.

[80] George C. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[81] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 229–243, Seattle, WA, October 1996.

[82] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus—an architecture for extensible distributed systems. In *Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993. (ACM Operating Systems Review, Vol. 27, No. 5).

[83] Gerald J. Popek and David R. Farber. A model for verification of data security in operating systems. *Communications of the ACM*, 21(9):737–749, September 1978.

[84] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.

[85] P. Richards. Timing properties of multiprocessor systems. Technical Report TDB60-27, Tech. Operations Inc., Burlington, MA, August 1960.

[86] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the Symposium on Security and Privacy*, pages 114–127, IEEE Computer Society, Oakland, CA, May 1995.

[87] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–53, 1996.

[88] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[89] John Rushby. Proof of Separability—A verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, Volume 137 of Springer-Verlag *Lecture Notes in Computer Science*, pages 352–367, Turin, Italy, April 1982.

[90] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-2, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1992.

[91] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[92] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI

International, Menlo Park, CA, March 1995. Also available as NASA Contractor Report 4673, August 1995, and issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification). Reprinted in [97, pp. 1–42].

[93] John Rushby. A foundation for security kernel verification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1995. Informal report.

[94] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Revised, July 1996. Available, with specification files, at `http://www.csl.sri.com/csl-95-10.html`.

[95] Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *ACM National Conference*, pages 404–410, Seattle, WA, October 1977.

[96] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[97] Roger Shaw, editor. *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*, Bruges, Belgium, September 1995.

[98] O. Sibert, P. Porras, and R. Lindell. The Intel 80x86 processor architecture: Pitfalls for secure systems. In *Proceedings of the Symposium on Security and Privacy*, pages 211–222, IEEE Computer Society, Oakland, CA, May 1995.

[99] Andrew Simpson, Jim Woodcock, and Jim Davies. Safety through security. In *Proceedings of the Ninth International Workshop on Software Specification and Design*, pages 18–24, IEEE Computer Society, Ise-Shima, Japan, April 1998.

[100] Andrew Clive Simpson. *Safety through Security.* PhD thesis, Oxford University Computing Laboratory, 1996. Available at `http://www.comlab.ox.ac.uk/oucl/users/andrew.simpson/thesis.ps.gz`.

[101] *Sixteenth ACM Symposium on Operating System Principles*, Saint-Malo, France, October 1997. (ACM Operating Systems Review, Vol. 31, No. 5).

[102] Oliver Spatscheck and Larry Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd Usenix Symposium on Operating*

Systems Design and Implementation (OSDI), pages 59–72, New Orleans, LA, February 1999.

[103] Christopher Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. In *Fault Tolerant Computing Symposium 28*, pages 218–227, IEEE Computer Society, Munich, Germany, June 1998.

[104] Jonathan T. Throstle. Modeling a fuzzy time system. In SSP'93 [46], pages 82–89.

[105] Paulo Veríssimo, José Rufino, and Li Ming. How hard is hard real-time communication on field-buses? In *Fault Tolerant Computing Symposium 27*, pages 112–121, IEEE Computer Society, Seattle, WA, June 1997.

[106] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[107] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, Asheville, NC, December 1993. (ACM Operating Systems Review, Vol. 27, No. 5).

[108] D. G. Weber. Formal specification of fault-tolerance and its relation to computer security. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 273–277, Pittsburgh, PA, May 1989. Published as ACM SIGSOFT Engineering Notes, Volume 14, Number 3.

[109] Doug G. Weber. Fault tolerance as self-similarity. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 2, pages 33–49. Kluwer, Boston, Dordecht, London, 1993.

[110] Charles B. Weinstock and John Rushby, editors. *Dependable Computing for Critical Applications—7*, Volume 12 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, San Jose, CA, January 1999.

[111] Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In Weinstock and Rushby [110], pages 287–300.

[112] John C. Wray. An analysis of timing channels. In SSP'91 [45], pages 2–7.

[113] Che-Fn Yu and Virgil D. Gligor. A specification and verification method for the prevention of denial of service. *IEEE Transactions on Software Engineering*, 16(6):581–592, June 1990.

Partitioning System Requirements and Architecture

David S. Hardin
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA 52498 USA

Abstract

This paper describes requirements and architecture for an operating system and associated hardware for avionics applications that provides confinement of programs (herein referred to as partitions) in both space and time. The operating system consists of a kernel that executes in the most-privileged mode of the processor and a set of library functions that execute in 'user' mode and provide an application programmatic interface (API) to kernel services. Partitions are protected from each other in space by memory management hardware internal to the CPU, or by a specially developed partition management unit. Partitions are confined in time by periodic scheduling events driven off a nonmaskable interrupt. The operating system framework provides partition scheduling on a strict periodic time slice basis, with an additional period for asynchronous event handling, similar to the approach described in the IEEE 1394 'FireWire' standard. Thus, both time-triggered and event-driven requirements are met, with a straightforward extension to distributed systems. Interpartition communications are provided by a set of message-passing API's.

1 What is Partitioning

Partitioning is the name given to the ability of a single avionics computer system to execute multiple concurrent, yet noninterfering, application programs. Such a system provides confinement both in space, through memory protection, and time, through periodic partition switching. Partitioning is distinguished from its close cousin, multiprogramming, chiefly by the fact that partitions lack the full semantics of, say, a Unix process. A partition is, in effect, a handy, rigid confinement vessel for threads of control and data.

There are two ways to produce a partitioning system. In the first scenario, applications that previously executed on separate CPU's are integrated onto a single processing site. In the second scenario, a previously monolithic program is split into partitions of differing criticalities, again executing on a single processor. The basic motivation for the first scenario is to save on hardware and associated costs (one powerful CPU versus several less powerful ones); the primary motivation for the second scenario is to save on verification, validation, and certification costs.

Partitioning operating systems have begun to appear in commercial avionics, most notably in the Boeing 777 AIMS, a large Integrated Modular Avionics ('cabinet') system [6] whose time-slicing schedule is driven by the [2] data bus. However, we see applications for partitioning operating systems all the way down to the traditional avionics 'black box' LRU level.

2 The Need for Safe, Secure Partitioning

The need for provably secure and safe hardware encapsulation mechanisms is increasing for both military and commercial avionics products, as more functions are integrated onto fewer high-performance processing sites. The basic technological trend driving partitioning is that available processing cycles per unit time are increasing faster (according to Moore's Law) than the demands of traditional avionics functions. (However, it is becoming increasingly difficult to procure the latest CPU technology for the extended environmental conditions required by avionics, so there may be a limit to how much functionality can be integrated.)

We foresee the use of verified hardware encapsulation both in the security realm, for example, in red-black separation, and in the safety realm, through the increasing use of 'brickwall' partitioning systems. Brickwall partitioning allows multiple, noninterfering applications of potentially differing verification levels to share the same processor; a formal basis for this partitioning would ease verification and certification burdens significantly.

In both the security and safety arenas, the sheer quantity of code in today's avionics functions calls out for hardware-and-kernel-enforced partitioning between unrelated software modules. In this way, reverification of unchanged components

can be reduced to confirming the interactions between the changed and unchanged components.

Formally verified hardware encapsulation techniques have widespread applicability that encompasses nearly the entire product line of Collins' civil and military avionics, radio, and global positioning systems.

2.1 Application to Secure Systems

One particular application area for formally verified hardware encapsulation technology in the security domain is the Global Positioning System Selective Availability Anti-Spoofing Module (SAASM). SAASM will be the next generation Precise Positioning Service security module, incorporating GPS Selective Availability, anti-spoofing, and electronic keying functions. The SAASM system architecture specifies separate red, yellow, and green program and data spaces for an application microprocessor, with a precise set of rules concerning access. Access control will require both hardware and software for encapsulation.

Collins GPS engineers are concerned about the effects of cache and scratchpad memory on separation for SAASM, as well as security flaws of off-the-shelf CPU's and DSP's.

Formally verified hardware and software encapsulation technology would provide a significant advantage for a military electronics supplier during security audits. In addition, proofs of correctness are beginning to be required of military contractors, for example, MOD-055.

2.2 Application to Safety-Critical Systems

2.2.1 Integrated Modular Avionics

Advances in high-speed microprocessor computing technology have recently provided the opportunity to reduce flight deck space, power, interconnect, and cost by "integrating" multiple avionics applications on a single processing site that shares resources such as memory. The most notable example of this is the Integrated Modular Avionics (IMA) architecture [6] developed by the AEEC Subcommittee for Systems Architecture and Interfaces [1] and the related APEX standard for interfacing application tasks to the executive operating system [3]. Integrated avionics is also becoming common in the military market.

Future designs will be even more integrated, especially as hydraulic and mechanical actuation systems are replaced by electronics and photonics. However, the ability to mix applications of different criticality levels on a single processing site also offers enormous benefits even for today's federated (single-function) systems by providing a means to concentrate verification on the most critical portions of a single application.

Issues that must be addressed when integrating previously federated applications include:

- Ensuring the isolation of applications that share common resources, such as the processor or memory, so that a faulty application cannot interfere with other applications. In federated systems, this isolation is provided through physical separation of the processing sites and resources.
- Providing reliable channels for communication between applications. In federated systems, such communication is provided by physical buses with well-defined protocols.
- Responding to external inputs or controlling external outputs within fixed time limits. In federated systems, each processing site performs a specific function designed to ensure that these limits are met. In integrated systems, the delays introduced by the other applications must be taken into account.

In the simplest terms, what has been demonstrated safe in federated systems must be shown safe in integrated systems. For example, in integrated systems a central problem is guaranteeing that each application receives its allocation of processing time and memory space and that corruption by other applications is inhibited or "brickwalled". In this sense, applications are still federated even though they reside on the same processing module. What architectural and kernel executive design features guarantee that this brickwalling occurs? How can these features be verified?

This is not a simple problem. For flight-critical applications, it is not clear how to verify the mechanisms for partitioning to the necessary levels of confidence. Testing alone is well known to be insufficient to ensure the levels of reliability needed [5, 9]. Reviews and inspections are invaluable in improving the reliability of digital systems, but are inadequate for the levels of assurance required here. Testing techniques are especially limited in the area of real-time programming where errors arise that are not repeatable due to different timing sequences.

Formally verified encapsulation technology, we believe, provides a means to assure that these federated system properties continue to hold in an integrated avionics environment, and thus will be of great value to designers of integrated modular avionics.

2.2.2 Federated Avionics

As noted above, partitioning can be applied to both integrated modular avionics and federated avionics. In a typical federated avionics application, for example, an EFIS (Electronic Flight Instrumentation System), partitioning could be used to segregate software into flight-critical and non-flight-critical functions, with kernel-mediated messaging between the partitions. This has several advantages:

1. Critical code can be isolated from the rest of the system. This isolation reduces intermodule interactions, thus enhancing safety, and makes verification of the smaller critical partition easier.

2. Application code is isolated from the hardware by the kernel [13, 14]. The hardware interface is thus concentrated in one piece of highly trusted code, again enhancing safety.

3. Verification burden is decreased, as not all modules need be certified to the highest level of criticality.

In a typical sensor application, for example, a GPS sensor, partitioning could be used to integrate BITE (Built-In Test Equipment) functionality, heretofore located on a separate maintenance processor, onto the main processor. This saves on both non-recurring and recurring hardware costs, without forcing BITE code to be developed and verified to flight-critical level.

In general, formally verified hardware encapsulation technology would provide a significant advantage for an avionics supplier seeking FAA certification of a safety-critical system incorporating partitioning. Indeed, formal verification of this technology would even permit its use in such ultra-safety-critical systems as autopilots. Formal methods have been recognized as an 'alternate method' for software development under RTCA (Requirements and Technical Concepts for Aviation) DO-178B [12], the software development standard for civil avionics.

2.2.3 Application to RTCA SC-182 Avionics Computing Resource

The technologies developed during this project could be transferred to the civil aviation industry as a whole if they are made part of the RTCA SC-182 standard. SC-182 is an effort to specify the Minimal Operating Performance Standard (MOPS) for a generic Avionics Computing Resource (ACR). The ACR is a set of general-purpose hardware and operating system software that supports brickwall partitioning, and can be 'pre-certified' for use in any number of avionics applications, such as FMS or EFIS. A generic ACR with a formal basis would be a very powerful building block.

3 Partitioning System Requirements

In setting down the requirements for a partitioning system, one must consider characteristics of both the hardware and software infrastructure, and capture those elements that are truly important for the preservation of partitioning. It is easy to omit crucial details in such an informal analysis; thus, the requirements formulated herein should not be considered sufficient. However, it is hoped that this domain-oriented analysis will guide, and indeed lend an additional degree of relevance to, the formal work to follow.

First, some definitions are in order:

- *Gong*: The fundamental partition switch interval. Abbreviated Tg.
- *Kernel*: Privileged, trusted software that manages partitions.
- *Partition*: A set of application tasks allocated to a single address space which is confined in memory and execution time by the system hardware and software.
- *Processor*: The computational hardware upon which the kernel and partition software executes.

Partitioning can be divided into space and time components; thus, we address the space and time requirements separately. But first, we must describe the computational environment for a reasonable avionics partitioning system, then establish requirements for the initialization of a partitioning system.

3.1 System Environment Requirements

[Requirement E1] No software shall disable system reset.

[Requirement E2] No processor execution may take less than some finite time interval, called the CPU cycle time, Tcpu.

[Requirement E3] Asynchronous interrupts may occur at any time, but will only be recognized on multiples of the CPU cycle time, Tcpu.

[Requirement E4] Failure to handle an interrupt before the next occurrence of that interrupt shall be detected.

[Requirement E5] Only the kernel shall handle processor interrupts.

[Requirement E6] The partition switch interval, designated the gong interval, Tg, shall be constant.

[Requirement E7] No partition shall be able to detect the presence of any other partition on the same processor.

[Requirement E8] No partition shall have direct access to peripheral hardware; all access to such peripherals shall be through the intermediation of the kernel.

[Requirement E9] No partition shall be able to execute privileged processor instructions.

[Requirement E10] The memory system shall be of a fixed absolute size, M; no secondary paging storage exists.

3.2 Initialization System Requirements

[Requirement I1] Control shall pass to the kernel after system reset.

[Requirement I2] No partitions shall be assumed to exist at system reset.

[Requirement I3] Only the kernel shall create and schedule partition memory spaces and execution time slots.

[Requirement I4] The maximum number of active partitions per processor shall be a fixed value, Np.

[Requirement I5] The kernel shall not schedule partitions such that the sum of the partition execution intervals (the Tp) and the maximum kernel execution time, Tk, shall exceed the gong interval, Tg.

[Requirement I6] Partition event handling shall not exceed a fixed interval, Te, expressed as a fraction of Tg.

[Requirement I7] The kernel shall be entered upon the occurrence of any nonrecoverable partition execution error.

[Requirement I8] Only the kernel shall terminate the execution of a partition.

[Requirement I9] The execution schedule of a partition shall be a function of its Tp, the global constants Tg, Tk, and Np, and the Tp values of the partitions active when it is created.

3.3 Space Partitioning System Requirements

[Requirement S1] Each partition shall have a designated memory space.

[Requirement S2] All the code and data for each partition shall be contained within its designated memory space.

[Requirement S3] No partition shall be permitted to read outside its designated memory space.

Requirement S4] No partition shall be permitted to write outside its designated memory space.

Requirement S5] No partition shall be permitted to change any partition's memory space boundaries or permissions.

[Requirement S6] The kernel shall have unrestricted access to partition memory spaces.

3.4 Time Partitioning System Requirements

[Requirement T1] Every active partition p shall be guaranteed a minimum execution time allocation, Tp, per gong interval.

[Requirement T2] No partition shall be permitted to exceed its execution time allocation per gong.

[Requirement T3] No partition shall be permitted to change any partition's time allocations.

[Requirement T4] No partition shall be permitted to interfere with the partition gong.

[Requirement T5] If the kernel's maximum execution time per gong, Tk, is exceeded, a system reset shall occur.

3.5 Interpartition Communications System Requirements

Given that we do not wish partitions to be directly aware of each other's presence for reasons of both isolation and incremental verification/validation (see Requirement E7), it makes no sense to speak of peer-to-peer interpartition communications. However, partitions must be able to transmit and receive information.

We take the approach of providing labelled, broadcast information flow; this is typical of avionics buses (e.g., [4]). To transmit a message, a partition first opens a logical device (managed by the kernel) for writing, designated by its device number or label. The partition then writes to the logical device using a kernel service (provided via a kernel API), and closes the logical device when it is no longer needed.

On the receive side, a partition indicates interest in a particular label by opening the appropriate numbered device for reading. A thread in the partition then typically blocks on the kernel read service, and is awakened when there is data on that label. A polling loop can also be used, if one can stand the overhead.

The kernel mediates all communications, from enforcing read/write policies on labels to ensuring that messages are delivered to all interested partitions. The kernel is also responsible for any fault-tolerance operations, such as voting. In the fault-tolerant model, communications may require several rounds [15]; partitions should not be concerned with this, other than the latency which such rounds introduce (which can be expressed simply in terms of time, without reference to any particular mechanism).

Since we do not assume a time-triggered model; thus, the kernel must assure that all messages have enough 'header' information to identify them. Typically, one or two bytes of label is all that is required, as all messages are broadcast, and thus do not require source and destination addresses in the headers.

Certain labels may be persistent; that is, reading the message does not destroy it. Persistence management is a system responsibility; the persistence attribute of a given label may not be changed by partition action.

[Requirement C1] No partition may directly communicate with another; communication occurs via kernel mediation.

[Requirement C2] All messages shall be labelled.

[Requirement C3] The kernel shall ensure that pending messages for a partition are the most current at the beginning of the partition's time slice.

[Requirement C4] A partition may read a label only with the permission of the kernel.

[Requirement C5] A partition may write a label only with the permission of the kernel.

[Requirement C6] A partition may not change the persistence of a given label.

4 Verification, Validation, and Certification of Integrated Applications

The verification, validation, and certification approach that we envision for the use of a partitioned operating environment is a straightforward extension of current avionics practice.

4.1 Demonstration of Excess Space and Time Capacity

Avionics products delivered to the government or to a commercial airframe customer must demonstrate the existence of 'spare capacity' in both space and time (usually expressed as a percentage of the delivered capacity). This leaves room for future growth, and has the side effect of reducing the overall 'stress' on the system. Spare capacity is usually demonstrated by various measurement and analyses of the final product.

It is fairly easy to demonstrate that excess space capacity exists: the code size is known at link time, and since most avionics systems statically allocate data memory, the RAM allocation is also readily determined. (Additional instrumentation of the heap may also be used to show that heap allocation never exceeds a fixed limit.) Demonstrating excess time capacity is a bit more complex, and can require analysis of the worst-case execution path of a given application; this is complicated by such real-world concerns as interrupts, asynchronous messaging, and real-time executive scheduling. A typical measurement approach is simply to monitor the amount of time spent in an 'idle' or 'background' task during execution of what the customer and developer agree is a realistic worst-case processing load.

In a partitioned system, the space and time 'fences' are set up in advance by the system integrator. The (formally verified) partitioning hardware and software ensures that no space or time allocations are exceeded; thus, excess space and time capacity are very readily demonstrated.

4.2 Verification and Validation

Avionics product testing typically addresses two separate concerns: verification ('Did we build the thing right?') and validation ('Did we build the right thing?') Verification testing is normally consumed with forcing all paths through the design in 'white box' fashion in order to show that all paths perform as expected and can indeed be reached. Validation testing, by contrast, determines whether the system performs properly from an external ('black box') point of view via a series of functional tests.

Partitioning can aid both verification and validation testing. In the area of verification, partitioning restricts the state space visible to any given application partition; this reduction in the state space should yield commensurate verification cost reductions. In addition, the separation of application and kernel tasks reduces the amount of hardware-dependent code in any given partition, allowing for portability of both the code and its associated verification tests. This is similar to the benefit derived from developing applications for modern, standards-conforming operating systems.

Validation testing will benefit in particular from partitioning in the area of integration testing. Since the partitioning system guarantees non-interference, par-

titions can undergo validation testing independently, both simplifying testing and relieving test schedule pressure. In a typical validation testing regime, testing would occur in an n-partition environment, where only the partition under test is actually running. The other partitions' time slices would be occupied with background processing, or with test stimulation routines. Alternately, multiple versions of a given partition can be run in the other partition time slices, and their outputs compared for regression testing purposes. In any case, the requirements of Section 3 guarantee that the partition will behave identically when integrated with other application partitions.

4.3 Certification

There have not been enough FAA and JAA certifications of partitioning systems for any sort of standard procedures and expectations to have developed. Suffice it to say that commercial avionics suppliers will find partitioning much less cost-effective if they cannot certify partitions separately. Formally verified partitioning should aid in the attainment of this goal.

5 A Formally Verifiable Partitioning Architecture

The following partitioning architecture is offered as a candidate for satisfying the requirements of Section 3, and is one that can be formally specified and verified.

In developing this architecture, we weighed the merits of time-triggered versus event-driven systems [8]. On the one hand, time-triggered systems are easily specified and reasoned about, as they require only periodic time slicing of partitions, and do not allow asynchronous event handling. On the other hand, the vast majority of real-world systems are to a large extent event-driven, especially those systems that feature operator interaction. (Indeed, interactive performance has been a significant problem in the time-triggered systems developed to date.)

In the end, we developed a hybrid architecture, one that allows for asynchronous events, but only a maximum number per gong. This hybrid approach allows us to address real-world environments more naturally, while also allowing the time-triggered architecture as a degenerate case (i.e., by setting the maximum number of event handlers per gong to zero).

5.1 Combining Time Slicing and Event Handling

As in a traditional time-triggered architecture, the hybrid architecture guarantees that up to a maximum number (Np) of partitions will be scheduled on a time-slice basis per gong, with a time slice of Tp. (An equal time slice interval per partition is not required, but is typical for time-triggered systems, and makes the following

discussion a bit easier.) Unlike a time-triggered architecture, however, each time slice is followed by an event handling interval, Te. (It is expected that Te o/ Tp in practice.) A maximum number, Ne, of asynchronous events may be handled in Te (one event per partition, in all likelihood), with a resulting latency of Tp. As the number of partitions Np increases, this latency compares favorably with the latency Np * Tp of a time-triggered system, that supports only polling.

5.1.1 Restrictions on Event Handling

Given that event handling is restricted to the interstices between partition time slices, one might expect that event handling might be quite limited. This is indeed the case; we envision that event handling would be restricted to vectoring to the interrupt handler, then transferring a data value from, say, a hardware peripheral, to a FIFO in the appropriate partition's data space. (NB: Here we have a weak coupling to the fact that there are a maximum of Np other active partitions possible in the system. If a partition needs to process j events per Te, each of which results in a datum being read into the partition's memory space, then the partition's event handling FIFO would need j * Np entries in order to avoid FIFO overrun. However, this does not appear to be an onerous situation, as the verification scenario currently envisioned requires that the maximum value Np is known.)

Note that for efficiency and hardware interfacing reasons, event handlers are executed by the kernel on behalf of the partitions; it is expected that any such handler would have to be simple in order to meet the kernel level verification requirements.

5.1.2 Relationship to IEEE 1394 (FireWire) Bandwidth Reservation

This hybrid scheme bears a strong resemblance to the various sporadic task scheduling approaches found in the literature (e.g., [10, 7]). However, this theme is not restricted to process scheduling. It is in fact similar to the bandwidth reservation scheme of the IEEE 1394 (FireWire) bus standard [16, p. 24]:

> Using time-division multiplexing, the cycle master allocates bus resources among competing nodes in a systematic fashion. One of its key functions is to transmit a timing message called a cycle start at regular intervals, generally occurring every 125 usec.
>
> The first portion of the bus cycle is available for isochronous data transmission. Since at least 20 percent of the cycle must be reserved for asynchronous traffic, the isochronous portion of the cycle may not exceed 80 percent, or 100 usec, in the case of 125-usec cycles. Asynchronous data fills the remaining portion of the cycle.
>
> On every cycle, nodes needing to transmit data, whether isochronous or asynchronous, must vie for control of the bus. An arbitration process

ensures that only one node transmits at a time and that all have fair access to the bus.

The nodes with reserved isochronous channels arbitrate first ... The process continues until all nodes wishing to transmit isochronous data have done so...

Next ... asynchronous arbitration occurs ... In addition, to give all nodes equal access, each node is allowed to transmit only once during the asynchronous portion of the cycle, also known as the 'fairness interval'.

Although the hybrid partitioning scheme and IEEE 1394 were developed completely independently, if one substitutes 'scheduling' for 'arbitration', 'gong' for 'cycle start', 'partition time slice' for 'isochronous transmission', etc., it can be seen that the hybrid architecture is quite similar to the FireWire approach. The main difference is that the hybrid architecture allows asynchronous events in between each time slice, whereas FireWire allows an asynchronous event interval only after all the isochronous transmissions have occurred. This resemblance is reassuring, and provides a rather obvious pathway to distributed systems, which is scheduled to be studied later in the contract.

5.2 Partition Data Required for Hybrid Architecture Scheduling

How does one determine whether a partition's processing requirements will be met by the hybrid architecture? First, the provider of the partition must indicate the periodic (time slice) processing requirements for the partition (Tp execution per Tg), the event handling rate, the event handling latency that can be tolerated, and the event handling duration. Then, given the maximum number of partitions, Np, the gong interval, Tg, and the event handling interval Te, the system integrator can determine whether the partition's time requirements can be met. The space requirements are easily met; one need only determine that the total memory load does not exceed M. Note that, as discussed in Section 4, the information required is no more extensive than that required for verification and validation of federated avionics.

5.3 System Design Using the Hybrid Architecture

In order to show the hybrid architecture at work, consider the following example:

Partition E is an Electronic Flight Instrument System, encompassing the 'basic T' instruments on one display. EFIS implementations are usually required to update the display at a 20 Hz rate, and typically poll their external interfaces at the same rate. EFIS systems must also respond to a variety of operator inputs, including brightness control.

Partition I is an Engine Indication/Crew Alerting System, or EICAS. EICAS 'synoptics pages' are typically required to update the display at a 5 Hz rate, and the EICAS receives engine data at the same rate.

Partition B is a background BITE (Built-In Test Equipment) partition; such functions are usually required to perform a complete check every 30 seconds, although they often complete before then.

Assume that typical kernel services (partition-to-partition context switches, etc.) take 200 microseconds (Tk = 0.2 msec). If we employ a rule-of-thumb that states that the kernel should not require more than 5interval of 4 msec, or 250 Hz. This is comparable to other operating system 'ticks'; Unix ticks, for example, are usually 100 Hz. A gong interval of 4 msec, however, is probably too short to perform one complete cycle of an avionics function. Let's see what Tg = 16.67 msec (60 Hz) would imply.

For our example above, each major cycle should complete every 20 Hz; this implies that Tg * Np = 50 msec, or Np = 3 for Tg = 16.67 msec. Assume that the duty cycle for event handling is 20partition execution, similar to [16]; this leads to an event handling interval, Te, of 3.33 msec every 16.67 msec. Also assume a similar 0.2 msec interval for each event to be handled; thus, in a 3.33 msec event handling time slice, up to 14 events may be handled (subtracting two 0.2 msec intervals for context switching). Assuming all partitions had at least one event handler, this would lead to a maximum Np of 14; systems currently envisioned by Collins do not exceed this number. More likely, each partition would have more than one event source. For the case of three event sources per partition, this would imply an Np of 4, which is still more than we need for the example.

This periodic event handling also implies 16.67 msec latency for all external events; this is probably tolerable for the example partitions above.

Of course, we are not counting any partition-specific event handlers in our kernel time allocation; the actual computation time available to application code is thus further restricted. Assuming Np = 3 and Te = 3.33 msec, the maximum time that any given application partition is allowed is Tp = Tg, Te = 16.67 msec - 3.33 msec = 13.33 msec every Tg * Np = 50 msec, for a partition rate of 20 Hz. The EFIS, EICAS, and background partitions, then, would run at a sufficient rate; and 13 msec is probably sufficient to perform each function, given a reasonably modern CPU. One final note: The problem of the computation of the 'optimum' gong interval is similar to the problem of selecting the Target Token Rotation Time (TTRT) for real-time token-passing datalinks [11]. Further exploration of this similarity will follow.

6 Conclusion

We have established partitioning system requirements from an avionics perspective, and have proposed a partitioning architecture that is both amenable to formal

verification and reflective of real-world computational environments.

References

[1] *ARINC Specification 651: Design Guidance for Integrated Modular Avionics.* Aeronautical Radio, Inc., Annapolis, MD, November 1991. Prepared by the Airlines Electronic Engineering Committee.

[2] *ARINC Specification 659: Backplane Data Bus.* Aeronautical Radio, Inc., Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.

[3] Avionics application software standard interface. Technical report, Airlines Electronic Engineering Committee, January 1997. ARINC Specification 653.

[4] Mark 33 digital information transfer system (DITS), part 1: Functional description, electrical interface, label assignments, and word formats. Technical report, Airlines Electronic Engineering Committee, September 1995. ARINC Specification 429P1-15.

[5] Ricky W. Butler and George B. Finelli. The infeasibility of experimental quantification of life-critical software reliability. In *SIGSOFT '91: Software for Critical Systems*, pages 66–91, New Orleans, LA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

[6] D. Hart. Integrated modular avionics. *Avionics*, August 1993.

[7] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the Twelfth IEEE Real-Time Systems Symposium.* IEEE, December 1991.

[8] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered? *IEICE Transactions on Information and Systems*, E76-D(11):1325–1332, November 1993. Institute of Electronics, Information, and Communications Engineers, Japan.

[9] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, pages 69–80, November 1993.

[10] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[11] N. Malcolm and W. Zhao. The timed-token protocol for real-time communications. *IEEE Computer*, 27(1):35–40, January 1994.

[12] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.

[13] John Rushby. Kernels for safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, chapter 13, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986).

[14] John Rushby. Critical system properties: Survey and taxonomy. Technical Report SRI-CSL-93-1, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993. Also available in *Reliability Engineering and System Safety*, Vol. 43, No. 2, pp. 189–219, 1994.

[15] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In Mario Dal Cin, Catherine Meadows, and William H. Sanders, editors, *Dependable Computing for Critical Applications—6*, Volume 11 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 203–222, Garmisch-Partenkirchen, Germany, March 1997. To appear in IEEE Transactions on Software Engineering.

[16] I. Wickelgren. The facts about firewire. *IEEE Spectrum*, 34(4):19–25, April 1997.

Part III

Strict Encapsulation: A Precise Characterization and An Application

Overview

One of the new and important results to come out of the work undertaken for this DARPA contract is the notion of *invariant performance*. The novelty and utility of this notion is that it precisely characterizes strict encapsulation in a way that is meaningful to developers as a contract between a partitioned system and its hosted applications, and as a property whose specification and proof assures that a given system demonstrates the requisite noninterference behaviors [17]. Our approach is illustrated through the formal specification and mechanical verification of the invariant performance of *Schultz*, a simple partitioning system consisting of the Rockwell Collins AAMP-FV microprocessor and a Partition Management Unit (PMU) [6].

Invariant Performance

Matthew Wilding
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA 52498 USA

Abstract

We describe a theorem about computer systems that support encapsulation of applications called the invariant performance theorem, explain why it is useful, and outline its formalization and proof.

1 Introduction

1.1 Overview

The DARPA "Formally Verified Hardware Encapsulation for Security, Integrity, and Safety" project is an effort to develop mathematical proofs of correctness to establish security, integrity, and safety all the way from the application software through the kernel software down to the design of a real microprocessor with related memory management or other protection mechanisms. The use of computer systems that support this kind of encapsulation is described in [2]. One challenge of this project is determining what should be proved about computer systems of this type; proposing such a theorem is the purpose of this paper.

Application builders have various concerns about their application's behavior. Will it do the right thing, will it run fast enough, will it process all its inputs, etc. Verification and validation of applications—especially safety-critical applications and applications with vital security properties—is very difficult. A great deal of analysis and testing is typically required to assure that an application works as needed. The prospect of hosting multiple applications on a single host is increasingly attractive because processors are increasingly powerful. However, this has the potential to make the verification and validation challenge more daunting since the potential for interaction between applications makes their behavior even more complex. Furthermore, all applications that share a host might need to be verified to the level appropriate for the most critical application running on the host.

One solution is to create a computing platform that hosts multiple applications in such a manner that they run independently and can be verified individually. The operating system support for this scheme can be localized in a small, trusted kernel. The purveyor of a given "partitioning" system is obliged to offer assurances of non-interference between applications using the system, a contract between the partitioning system and the hosted application. We will state one such contract and use machine-checked mathematical reasoning to ensure that a partitioning system fulfills it.

What contract should we formalize to simplify the application builder's verification and validation task? We address this question in later sections, but to begin we observe what is useful about such a correctness theorem:

> The correctness theorem provides a contract with the application developers that allows them to verify and validate the safety and security properties of their applications in isolation.

This section describes a useful theorem about systems that support partitioning, which we call the invariant performance theorem. Section 2 presents a statement of this theorem.

1.2 Virtual Machines

One approach to describing what an encapsulation-supporting computer system is required to do is to introduce the notion of an abstract, "virtual" machine. That is, to describe the operation of the system with respect to one of its partitions in terms of the operation of a more abstract computer system with only a single partition. A notable example of this approach is described by Rushby [5].

At the high end, the system could be conceived as a distributed one where the significant issues are those of controlling access to information and the communication of information between conceptually separate single-user machines. The fact that all users actually happen to share the same physical machine should be masked at this level.

The work described in [5] uses a virtual machine to motivate the appropriateness of a set of proof obligations for kernels of this kind. A more concrete representation of a virtual machine that is implemented by each of the partitions in a partitioning system is presented by Bevier in his KIT verification work [1]. KIT is an operating system kernel that enforces isolation and is proved using a mechanical theorem prover to provide to each running task resources consistent with an isolated, dedicated machine.

The use of a virtual machine allows a clean description of a notion of isolation, and it is appealingly intuitive. However, this kind of theorem is not quite the contract that the application developer needs in order to allow him to verify his application in isolation. If the application developer is unable to verify his application with respect to a particular, realized machine whose behavior he can predict then his verification burden will increase. Note in particular the complexity of the timing properties of such abstract machines: the virtual machine used as the specification for KIT has a "sputtering" clock whose "ticks" correspond to the execution of the partition containing the running application.

The use of a virtual machine such as used in the KIT specification would make the contract between the partitioning system and the application developer less useful since it is unclear how the application developer might verify his application with respect to such an abstract processor.

1.3 Invariant Performance

Rather than use an abstract machine to describe the partitioning contract, we will use a simpler, more concrete notion. A partition has invariant performance if its behavior is a function of its initial state and its inputs. We will provide as a partitioning contract that each of the partitions supported by our system has invariant performance, a theorem we call the invariant performance theorem.

The invariant performance theorem is in many respects a weaker claim than might be made using an abstract machine formulation of correctness since it does not

characterize "correct" partition behavior, only that the behavior is unchanged by the operation of other partitions. Note for example that an encapsulation system that does not execute any of its partitions satisfies the invariant performance although it is obviously not useful. Invariant performance is nevertheless the crucial correctness property of a partitioning system because it allows applications to be developed in isolation. The application developer who runs his application in a partition is assured by this theorem that his application will run identically when changes are made to other partitions.

Of course, other theorems about system performance are interesting and potentially useful contracts with application developers. This would be particularly true if one were to apply formal analysis to the verification of the applications hosted by partitions. For example, a minimum CPU allocation contract ("at least X CPU seconds every Y seconds") could aid the analysis of applications. We do not mean to suggest that invariant performance is the only interesting property that might be proved about a system that supports encapsulation, only that it is the crucial property for being able to develop applications in isolation using conventional verification and validation techniques.

We will discuss this theorem in more detail in Section 2 when we present its formalization.

1.4 Architectural Implications of Invariant Performance

The invariant performance theorem requires a very strong notion of separation between partitions. How realistic it is to expect an encapsulation system architecture to implement this? There are two concerns: is it realistic to expect application developers to use it, and is it realistic to implement? We believe the answer to both these questions is yes. The "boxcar" architecture is a partitioning system architecture [2] that is consistent with the notion of invariant performance. This architecture allocates the CPU among the active partitions with a round-robin schedule. Inputs are sensed between partition executions, thereby achieving predictable partition scheduling (since partition execution is not interrupted by input handling) with relatively little interrupt processing latency (since inputs are sensed at every task switch).

It appears realistic to implement the boxcar architecture so as to satisfy the invariant performance theorem. The architecture's uncomplicated approach to scheduling is crucial and contrasts sharply with, say, an architecture with multiple priority levels or immediate interrupt handling. We currently anticipate only two areas where we might use an implementation approach that we might not use if we were not building to fulfill such a strong separation contract, and both of these involve nothing eliminating a few microcycles of "slop" when the system swaps partitions. First, rather than a single signal to trigger partition swap we foresee a "double"

signal, the first causing the current partition to cease and the second to cause the succeeding partition to commence. This eliminates the effect on other partitions of unpredictable partition-switching signal latency caused by the non-interruptibility and variable execution time of processor instructions and kernel calls. Although this time is small since it is bounded by the slowest user-accessible kernel call, eliminating it allows us as partitioning system builders to make a far simpler and more usable contract with the application programmer. Second, we will ensure that an input is read on a partition swap if it arrives before a certain, predictable time. This eliminates the chance that one input's processing during a partition swap could determine whether another input is processed during the same partition swap or whether it is delayed.

The question of whether the boxcar architecture is capable of meeting the requirements of a particular application is perhaps the more challenging question since it cannot be answered absolutely. The ability of an architecture to support an application depends of course on the exact requirements, most notably the real-time requirements, of that particular application. Nevertheless this architecture appears to be a realistic approach to hosting multiple applications in a partitioned environment. The simple scheduling approach appears attractive for multiple applications and, as argued in [2], overhead incurred from supporting this architecture allows response times and CPU allocations for this system that are consistent with the requirements of a variety of avionics applications.

In short, although invariant performance is a very simple and quite constrained approach to sharing a processor between applications, we believe that the boxcar architecture suggests that it is a realistic.

1.5 Safety, Security, and Invariant Performance

Safety can be enhanced by a partitioning system because a faulty application is unable to compromise the effectiveness of another application running in another partition. Security can be enhanced by a partitioning system because it restricts the interaction of applications in different partitions. The concerns of application developers with respect to safety and security overlap to a large extent, but are not necessarily identical. An example that illustrates the difference between safety and security properties of partitioning systems involves cache associated with a swapped-out partition. If the values of the cache are accessible to another partition then security may be compromised, but this kind of leakage would presumably not effect safety.

The differences between safety and security are not apparent in systems that enforce invariant performance when there is no direct communication between partitions. Since invariant performance is the strongest possible notion of separation any system that provides it cannot hope for a stronger notion to ensure safety or

security. Consider the example above – the use of this kind of "leaked" information is not consistent with invariant performance since applications run identically regardless of the behavior of other partitions. What about systems that involve communication between partitions? Invariant performance is a very useful property when verifying those kinds of systems too. Let's assume that the output of partition A is used as input by partition B. We can analyze and test each of these partitions in isolation because we know they do not interfere with each other. Of course, we may wish to demonstrate other aspects of the behavior of the system. For example, we may need to demonstrate that partition B works when inputs arrival is no faster than some rate, and that partition A does not exceed that rate. Application of the invariant performance theorem allows us to decompose the task of verifying these properties.

Safety and security of a system are of course enhanced when the system is a high-integrity system. Fault-tolerance is an important consideration of partitioning systems. We have not yet investigated in detail how an architecture implemented to fulfill the invariant correctness theorem can be enhanced to withstand faults. However, we believe that simple, predictable execution will be more amenable to this kind of enhancement than more complex architectures. We may adopt a fault-tolerance approach similar to that in [6], where fault-tolerance is achieved using round-based communications. Communication can be isolated to the interregnum between partition execution in our model, thus separating to a large extent the fault-tolerance from invariant performance.

2 The Theorem

2.1 Formalization

This section presents a somewhat more formal description of the invariant performance theorem. The approach to processor modeling is inspired by several other projects. (See, for example, [8, 9].)

time is represented by a natural number. We think of it as the number of clock ticks of the processor. *value* is a 32-bit natural. A *channel* is a function of type $time \rightarrow value$. A *channel set* is a function of type $name \rightarrow channel$.

A value of type *processor state* represents the state of a processor. A processor model we have been experimenting with that is based on the AAMP-FV processor "macro" model [3] contains 8 fields: memory, page register, top-of-stack pointer, program counter, local environment pointer, user/exec flag, interrupt mask register, and interrupt state register.

$processor(s, i, o, ct, mt)$ is a function whose arguments are an initial processor state, a channel set representing inputs, a set of names representing some output channels, a time representing the "current" time, and a time representing the last

value of interest. *processor* returns a channel set whose domain is o and the domain of each channel in the range is $\{n : nat \mid ct \leq n \leq mtg\}$, where the values are consistent with the processor starting in processor state s with input channel set i. *partitioned*(n, s) holds when processor state s is executing with n partitions. This requires that

- the kernel code is loaded
- the kernel data structures include a valid memory map for n partitions
- the kernel data structures has input and output channels as associated with each of the partitions
- no channel is an output channel of more than one partition
- the interrupt mask partition switch ("gong" in the sense of [2]) interrupt is enabled

pmem(n, s) returns the memory associated with partition n in processor state s.
inputs(n, s) returns a set of names corresponding to the input channels of partition n in processor state s.
outputs(n, s) returns a set of names corresponding to the output channels of partition n in processor state s.
together$(i, s1, s2)$ holds when

- processor states $s1$ and $s2$ are executing in the same partition
- either the current partition is not partition i or the program counters of states $s1$ and $s2$ match

valid(in) holds when in is a channel set that includes the distinguished partition switch ("gong") interrupt and the gong channel values do not require too-fast partition swapping.

The invariant performance theorem is:

$$\forall(i, n, ct, mt : nat; s1, s2 : processor\ state; in : channel\ set) :$$
$$(i < n \wedge partitioned(n, s1) \wedge partitioned(n, s2) \wedge$$
$$pmem(i, s1) = pmem(i, s2) \wedge valid(in) \wedge$$
$$together(i, s1\ s2) \wedge inputs(i, s1) = inputs(i, s2) \wedge$$
$$outputs(i, s1) = outputs(i, s2))$$
$$\Rightarrow$$
$$processor(s1, in, outputs(i, s1), ct, mt) =$$
$$processor(s2, in, outputs(i, s2), ct, mt)$$

2.2 Proof Approach

We will prove this theorem using the mechanical proof system PVS [4, 7]. The chief advantage of developing a proof in this fashion is that the proof is unlikely to contain a mistake, since only the exploitation of a PVS soundness bug by the proof developer can lead to the false certification of a conjecture as a theorem. In this section we briefly outline the proof of the theorem.

The proof is by induction on the difference between the "maximum" time to be considered and the "current" time. The base case is trivial because the function *processor* returns a function with empty domain. A step is the computation that can occur between two instants that a partition swap could potentially begin, which can be a user-called kernel service request, the execution of some other kind of instruction, or a partition swap. We instantiate the invariant performance theorem with the state that results from executing an arbitrary state s one step and use this as an induction hypothesis. The induction is justified in part because a step requires at least one microcycle.

The bulk of the proof effort will be to establish that the hypotheses of the induction hypothesis holds. That is, that each of the assumptions described in the previous section hold after one step of the computation. The proof breaks into a case for each of the three kinds of step, which in turn breaks into cases for each instruction, each user-callable kernel call, and the partition swap. So, for example, we must show that the "read-from-input" kernel call does not falsify the *partitioned* predicate, that the "add" instruction does not disable the interrupt mask with respect to the gong interrupt, that the partition swap does not change the kernel code, and so on.

Once the (many) induction hypothesis hypotheses are proved, the theorem conclusion follows from the induction hypothesis conclusion. One of the lemmas that is needed here is that the execution of an instruction depends only on the state of the kernel and the partition, for example, instruction timing is such a function.

One interesting note about this proof: although our representation of time is very concrete—it relates back to the underlying processor clock—very little must be proved about instruction timings.

We expect that this will be a lengthy proof. The proof's complexity depends significantly on how complex the kernel turns out to be, but that complexity will be minimized. The proof will probably be more complex than the AAMP-FV microcode proof [3], since although there will probably be somewhat less code the behavior of that code is more complex. We are optimistic that with modest PVS performance and better proof engineering we will accomplish this with somewhat less overall effort than was required for the AAMP-FV effort.

References

[1] William R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.

[2] David Hardin. Partitioning system requirements and architecture. Delivered to SRI under contract, July 1997.

[3] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996. Also available under the titles *Formal Verification of an Avionics Microprocessor* as NASA Contractor Report 4682, July, 1995, and SRI Technical Report SRI-CSL-95-4, February, 1995.

[4] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[5] John Rushby. Proof of Separability—A verification technique for a class of security kernels. In *Proc. 5th International Symposium on Programming*, Volume 137 of Springer-Verlag *Lecture Notes in Computer Science*, pages 352–367, Turin, Italy, April 1982.

[6] John Rushby. Formal verification of an Oral Messages algorithm for interactive consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992. Also available as NASA Contractor Report 189704, October 1992.

[7] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[8] Matthew Wilding. A detailed processor model for verification of real-time applications. In *2nd IFAC Workshop on Safety and Reliability*. 1996.

[9] Matthew Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, pages 177–184, NASA Langley Research Center, Hampton, VA, September 1997. Available at `http://atb-www.larc.nasa.gov/Lfm97/proceedings/`.

Invariant Performance and PVS: A Statement of Task Isolation that can be Machine-Checked

David Greve Matthew Wilding*
Rockwell Collins, Inc.
Advanced Technology Center
Cedar Rapids, IA 52498 USA

*Much of the material in this report was published in Matthew Wilding, David Hardin, and David Greve, Invariant Performance: A Statement of Task Isolation Useful for Embedded Application Integration. In Charles Weinstock and John Rushby, editors. *Dependable Computing for Critical Applications—7*, Volume 12 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 287-300, San Jose, CA, January 1999.

Abstract

We describe the challenge of embedded application integration and argue that the conventional formal verification approach of proving abstract behavior is not useful in this domain. We introduce *invariant performance*, a formulation of task isolation useful for application integration. We demonstrate invariant performance by formalizing it in the logic of PVS for a simple yet realistic embedded system, outline its proof informally, and describe the machine-checked proof of an important sublemma.

1 Introduction

Integration of multiple real-time embedded applications onto a single processor is increasingly attractive because the capacity of computing devices continues to grow. The use of fewer devices reduces space and power consumption that can be very valuable in an embedded environment, and fewer device connections increase reliability. Greater integration can also simplify the development of fault-tolerant architectures.

Integration of applications poses daunting challenges as well, because integrated applications may interact. Applications that share computing resources can interfere with each other's *space* — values saved in memory by an application — and *timing* — the rate at which an application performs.

The system developer who wants to integrate an application with other applications has several concerns.

safety Can any other application in the system effect an application's performance to cause it not to meet its requirements?

security Can other applications in the system glean information that should be restricted to one application?

verification level Must each application be verified at the confidence level associated with the most critical application in the system?

verification completeness Has the verification of each application taken into account the many ways other applications might interfere with it?

These challenges can be met if the host computer system provides an encapsulation mechanism that separates applications so that they can be verified separately. This kind of mechanism, known as a "partitioning" system in the avionics community, allows not only the verification of integrated applications initially but can eliminate the need for reverification in future system configurations. An encapsulation mechanism must be no less trusted than the most trusted application in the system, so it is natural to turn to formal verification to gain a high level of confidence.

Operating system correctness statements in the literature have proved inadequate for application integration, so we have developed our own that we call *invariant performance*. In this paper we describe some previous work by others and introduce our notion of task isolation that allows separate verification of applications. We describe a small operating system and underlying hardware and a correctness statement that supports encapsulation useful for separate verification of integrated applications.

2 Verified Operating Systems

2.1 Verified Computer Systems

Machine-checked computer system proof has been used to build extremely reliable computer systems. Some examples of these involve compiled routines from the C string library targeted to the Motorola 68020 [5], microcode for the Motorola CAP processor [6], a stack of verified systems [3], verification of the "oral messages" algorithm [4, 10], code for some simple real-time systems [18], floating-point microcode [6, 15], a verified Piton [13] program [17], floating-point hardware [16], a simple scheduler [7], and partial microcode correctness of some Rockwell Collins microprocessors [11, 12].

Broadly speaking, each of these projects relates the execution of a model of a computer system with the execution of a more abstract model that describes the expected behavior of the system. This approach has also been used to establish aspects of operating system correctness. The most applicable example of the formal verification of an operating system is Bevier's verification of KIT [1, 2].

2.2 KIT

KIT ("kernel, isolated, tasks") is an operating system kernel that enforces partitioning among a set of user tasks. The kernel supports some simple communication services for interpartition messages and input devices. KIT is implemented with about 300 instructions of machine code for a machine called TM, a fictional machine with a simple von Neumann architecture. TM has some hardware support useful for building a partitioning operating system kernel, including

- a memory protection scheme involving base and limit registers,
- user and executive modes that effect instruction execution,
- input and output ports that provide a communication mechanism, and
- a clock that decrements after each instruction and generates an interrupt when it reaches 0.

KIT provides a set of user-mode tasks with the illusion that each is operating in isolation, except for the effect of interprocess communication. Each I/O device is associated with exactly one partition. Most of the verification work to establish partition isolation involves showing that KIT correctly maintains the state of the various tasks in the face of partition swapping and task execution, and that it does the right thing when there is communication with I/O devices or between partitions.

The correctness theorem relates a very detailed and realistic model of KIT executing on TM to an abstract model where tasks execute in isolation. The KIT correctness statement embodies in microcosm the appeal of formal methods. Rather than attempt to list all the things that could go wrong in a partitioned operating system (a hopeless task for all but the simplest of systems)

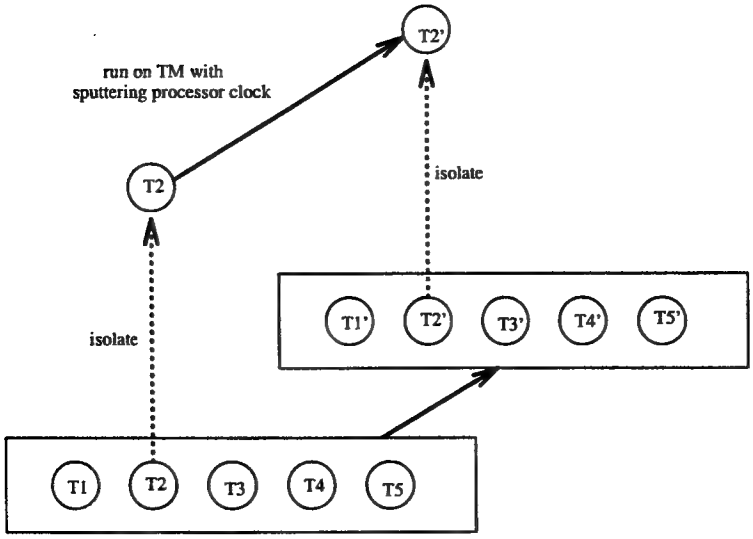


Figure 1: The KIT correctness theorem[2]

and show that nothing undesirable occurs, we demonstrate correct functioning of the system. Once this specification is proved we need not worry that we checked that a partition might overwrite another partition's memory space by, say, adding a value using an indirect address.

The KIT correctness theorem relates the execution of each partition running in isolation on an unloaded TM to its execution under KIT, and is illustrated in Figure 1. The partition running on the abstract machine used to specify KIT's behavior executes in spurts that correspond to the allocation of the CPU by the kernel at the more concrete level, and this relationship is formalized in the statement of the correctness theorem.

2.3 Invariant performance

The KIT theorem — and most of the other verification projects with which we are familiar — provides a service guarantee about a computer system. Real-time concerns are not considered in the KIT verification, but the approach used by KIT could be extended to provide guarantees about throughput and latency.

But these properties do not suffice to allow separate verification of integrated applications. *It is unrealistic to expect application developers to verify applications against an abstract, unrealized machine model, even if the abstraction arguably characterizes "good" system behavior.* Our specification dispenses with the definition of the abstract machine altogether. We will use machine-checked proof to relate the execution of an application on a realistic model of the computer system with the application's execution when other applications are different. This approach is suggested by Figure 2: partition **T** operates without regard to the operation of the other partitions. If two kernel-controlled

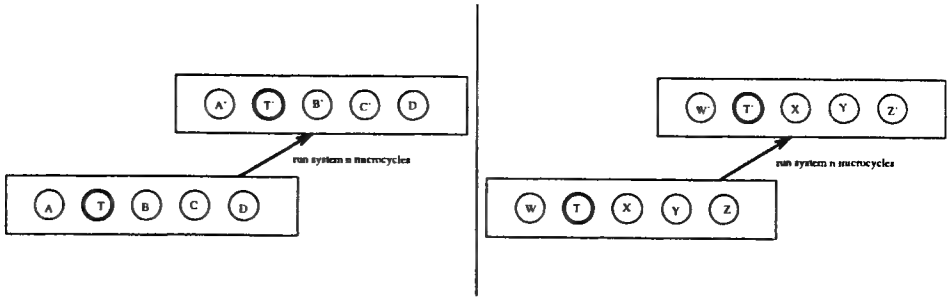


Figure 2: Invariant Performance

partition sets have identical initial kernel states and identical states for some partition **T** then the execution of the partition sets maintains the equivalence of the states of **T**.

The guarantee invariant performance makes to the application developer is that after his application is integrated with other applications it will work precisely as it worked before integration. If for example the developer tests his application in a partition he can rely on the system to work identically even if other partitions are used to host other applications. This can not be done, for example, with the KIT correctness statement: the developer does not know precisely how his application will run, only that it will run just like it would run on an abstract machine described by the correctness theorem. The abstract machine used in the KIT specification is like the concrete machine except that the clock sputters in a way that corresponds to CPU allocation under KIT.

We believe that invariant performance is the crucial property needed to allow separate verification of real-time, embedded applications since it provides complete time and space isolation. It also provides the strict separation needed to host applications that require isolation from each other in order to satisfy a security policy. It provides the application developer a development platform that will be unaffected by the application's integration, thereby allowing for independent verification of real-time, embedded applications.

3 An Example Application of Invariant Performance

We demonstrate invariant performance by developing its statement on a simple, concrete system. The system we model contains an AAMP-FV microprocessor [11] and a partition management unit ("PMU"). The PMU maintains memory isolation among the partitions and includes timers that allow temporal control of partitions by the kernel. The model of the AAMP-FV we use is adapted from the instruction-level AAMP-FV "macrolevel" model [11].

In this section we describe *Schultz*, a simple partitioning system, our formalization of invariant performance for Schultz, and an outline of the proof.

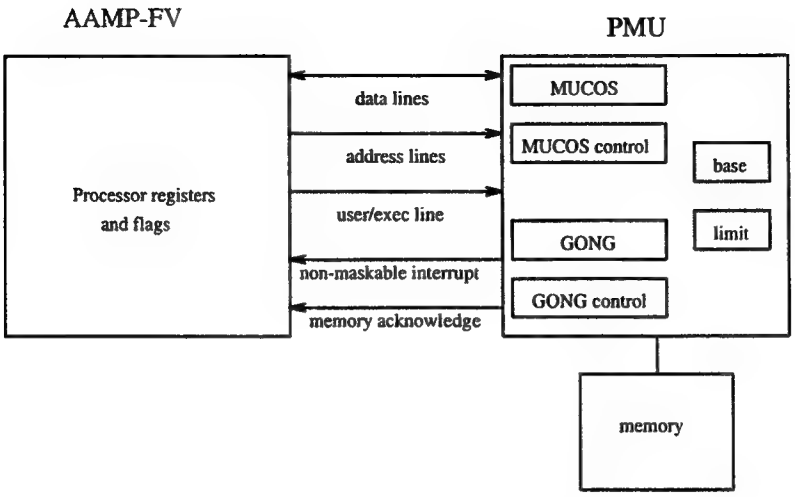


Figure 3: Schultz Hardware

3.1 Schultz Overview

Schultz supports a cyclic schedule of noninterfering partitions. There is no explicit provision in this initial model for kernel-mediated shared resources such as interrupt controllers. Each partition is allocated a predetermined amount of CPU time with no interruption and has its own memory space, thereby ensuring noninterference between partitions. The Schultz kernel is very simple since there is only one shared resource, the CPU, and is responsible for initializing the system and maintaining the partition schedule. An unusual aspect of Schultz is microcycle-accurate partition-starting to achieve strict temporal independence among partitions.

Figure 3 represents the Schultz hardware configuration. The AAMP-FV and the PMU interact in several ways.

- The accessible memory range is determined by the PMU using the processor user/executive line and base/limit addresses contained in memory-mapped PMU registers.
- For each processor memory transaction the PMU determines whether the location is currently accessible.
- A PMU timer, the *gong timer*, generates a periodic non-maskable interrupt ("NMI") to the processor.
- A PMU timer, the *mucos timer*, is used to generate a memory acknowledge signal to the processor when the processor writes a non-zero value to a particular memory address, the *mucos register*. As will be discussed below, this mechanism provides for strict temporal synchronization.

- Each PMU timer has a corresponding memory-mapped control register that determines whether the counter is active and, if so, the value to which the timer is set when it expires.

Our AAMP-FV model is a modification of the model of [11]. We have modified the model in order to simplify underlying proofs about code following the suggestions of [20], particularly the use of an interpreter style to facilitate code proof automation.

The Schultz PMU maintains information about the current partition executing on the processor, including how long its CPU allocation is, how much CPU time it has used, and the range of memory to which it has access. Expiration of the "gong" timer signals the end of the current partition's CPU allocation. The "mucos" timer signals the end of partition switch handling time and is used to time the start of a partition, thereby synchronizing the timing of partition execution to eliminate any effect of unpredictable interrupt latency — interrupts are only recognized on instruction boundaries — or early partition exits due to illegal instruction execution. The timers are free-running timers that are decremented each microcycle. When a counter reaches 0 it is reinitialized with a corresponding timer initialization value. A timer is shut off by setting the timer initialization value to 0.

The two PMU timers operate differently, reflecting how they are used. The gong timer is used to signal partition switching and generates an NMI when it reaches 0. The mucos timer is used to synchronize the start of a partition to the correct microcycle and delays the memory acknowledge signal on non-zero writes to the mucos register until the mucos timer reaches 0.

The PMU also maintains a base/limit pair for the currently operating partition that is used to restrict partition access to memory. When a user-mode instruction writes to memory outside the range of addresses between the base and limit pair it has no effect, and when an address from outside the range is read it returns a value of 0.

We express the model and its correctness conjecture in the logic of the PVS theorem prover [14] since we intend to prove that Schultz provides invariant performance, using PVS to machine-check the proof. The PVS formalization of the Schultz hardware is presented in Figure 4. We have not provided enough information for the reader to understand this model fully. (The subsidiary functions are defined in our model but not presented in this paper, and we do not attempt here to describe the semantics of the PVS logic.) We present Figure 4 to convey that we have modeled the Schultz hardware in detail to support reasoning about it.

4 The Schultz Kernel

We now introduce a partition switch signal handler that implements partitioning. The AAMP-FV assembly code is given in Figure 5. The loop initializes the PMU and schedules the next partition in the schedule.

```
step(system): system_state =
  LET conns = conns(system), s = processor(system), p = pmu(system) IN

  % run next instruction with protected memory and PMU-generated NMI
  LET s2 = step_processor(s WITH [(mem):=protectmem(memory(conns),p,conns),
                                  (intreg):=setint6(intreg(s), nmi(conns))],s) IN

  LET mem2 = IF um(s2) THEN
               restoremem(mem(s2),memory(conns),p,conns)
             ELSE mem(s2) ENDIF IN

  % calculate time of instruction execution
  % (mucos register active when in exec mode and timermod is positive)
  LET itime = step_time(s,mucosset(mem2),val(mucostimermod(p))>0,
                        val(mucostimer(p))) IN

  % update connections to reflect PMU/processor operation
  LET conns2 = (# (nmi) := (val(gongtimermod(p))>0 AND val(gongtimer(p)) <= itime),
                  (memory):=resetmucos(mem2),
                  (um) := um(s2),
                  (itime) := itime #) IN

  % update PMU state
  LET p2 = step_pmu(p,memory(conns2),itime) IN

    make_system (s2, p2, conns2)

protected_system(system,cur,finish): RECURSIVE system_state =
  IF finish<=cur
   THEN system
   ELSE
     LET system2 = step(system) IN
       protected_system (system2, cur + itime(conns(system2)), finish)
   ENDIF
  MEASURE max(0,finish-cur)
```

Figure 4: Schultz Hardware Formalization

```
ELOOP:  LIT16 OFFFFh                ;
        ASN24 exec_code             ;

        REF24 length                ; if at end of schedule, reset pointer
        REF24 curr                  ;
        SUB                         ;
        SKIPT8 rpart                ;
        LIT4 0                      ;
        ASN24 curr                  ;

rpart:  LIT24 psds                  ; store psd table location
        REF24 curr                  ;
        DUP                         ; while we've got it, set PMU num reg
        ASN24 partn                 ;
        LIT4 4                      ; multiply curr pointer by 16 to get offset
        SHL                         ;
        LIT4 0                      ; make single word a double word
        ADDD                        ; calculate PSD pointer for next partition

        DUP                         ; set PMU base and limit from PSD
        LIT32 8                     ;
        ADDD                        ;
        DUP                         ;
        REFDA                       ;
        ASND24 breg                 ;
        LIT32 2                     ;
        ADDD                        ;
        REFDA                       ;
        ASND24 lreg                 ;

        LIT4 1
        LIT24 mreg
        ASNA                        ; write to mucos register

        LIT4 7                      ; clear pending interrupts - for example,
        CLRINT                      ; an NMI that occurred since illegal instr

        USER                        ; start user partition

        ASN24 exec_code             ; save returned value - could be partition
                                    ; signal or illegal instruction

        REF24 curr                  ; increment schedule pointer
        LIT4 1                      ;
        ADD                         ;
        ASN24 curr                  ;

        LIT16 ELOOP                 ; calculate skip value to loop


        SKIP                        ;
```

Figure 5: Partition Switch Kernel Code

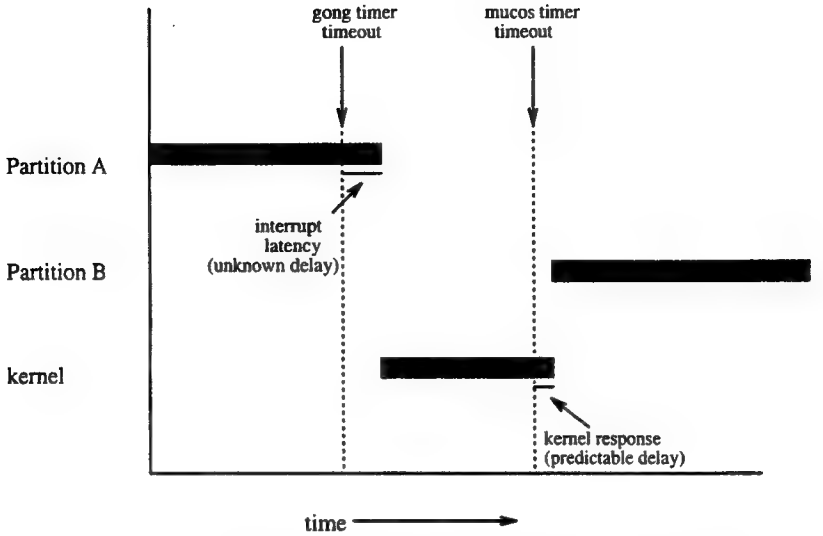


Figure 6: A Typical Schultz Partition Switch

The scheduling code maintains several data structures. A schedule of partitions is assumed at symbolic location `psds`. The schedule is saved using a list of length `length` of process state descriptors ("PSD"s), each of which represents a partition. An AAMP-FV PSD is 8 double words containing processor state values PC, TOS, LENV, and PAGE and 4 unused locations. Schultz uses 2 of the spare PSD double words to maintain the base and limit values for each partition.

The symbolic value `curr` is the number of the current partition in the schedule. Each time a partition switch occurs we increment `curr`, thereby advancing to the next partition in the schedule. The schedule is a simple, cyclic schedule, so when the value of `curr` reaches `length` it is reset to 0.

The partition switch handler uses the AAMP-FV USER instruction to start the next partition. Before executing the USER instruction, however, the handler reads the mucos register and clears pending interrupts.

Strict time partitioning requires that partition scheduling be unaffected by what individual partitions do. This requires some effort in the face of interrupt latency, since interrupts like the partition swap interrupt are only recognized on instruction boundaries. The partition switch handler uses the PMU mucos register to guarantee strict time partitioning. Before starting a partition the mucos register is referenced. The memory transaction acknowledgement signal for the mucos register address is delayed until the mucos timer expires. Thus, the start of the next partition is delayed until a predictable time, as illustrated in Figure 6.

The handler also addresses another scenario: it is possible that the previous partition ended not because of a partition switch signal but rather due to the partition's execution of an illegal instruction, as illustrated in Figure 7. If this

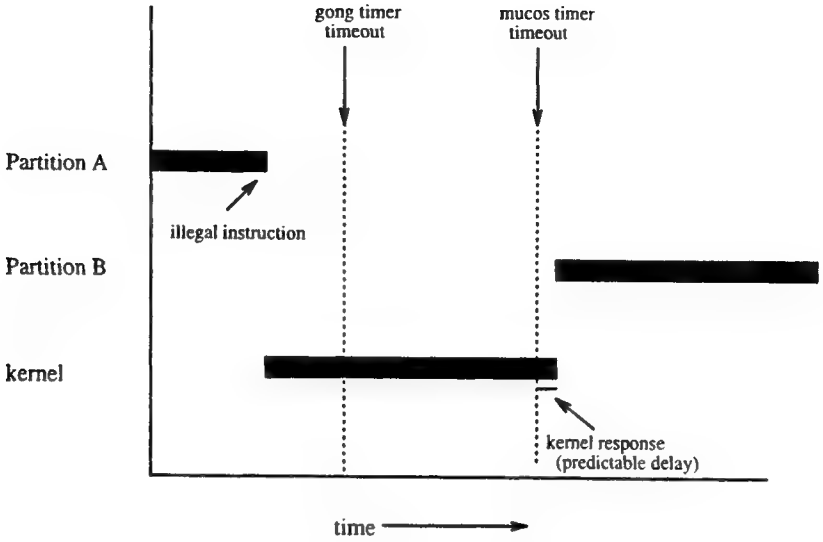


Figure 7: Partition switch after illegal instruction

occurs then during execution of the handler the partition swap signal will be generated; and since the handler runs in executive mode the partition switch interrupt will be pending. If this happens it will occur before the mucos timer times out (since the mucos timer is designed to time out after the gong timer) so the partition switch interrupt will be pending after the mucos synchronization. The kernel executes a CLRINT instruction to clear this pending interrupt, thereby ensuring that the next partition receives its entire CPU allocation.

5 Schultz Invariant Performance

Figure 8 presents our PVS formalization of invariant performance for Schultz. Given two good initial Schultz states that have equal length schedules with a valid schedule element i that has identical base/limit values and processor memory that is identical for that region, then running the system will yield a state with identical memory in that region starting in either state.

The conjecture has four universally-quantified variables: `sys1` and `sys2`, initial system states, `i`, a schedule element, and `fin`, an ending time. The hypotheses of the conjecture restrict the applicability of the conclusion to "reasonable" initial states and schedule elements — the initial states have the kernel loaded, the schedule element must have identical corresponding memories in the initial states, the schedule element must actually be in the schedule, etc. The ending time at which the memories must correspond has no restriction — the conjecture must hold at all times during execution of the system.

```
schultzip: LEMMA
  FORALL (sys1,sys2: system_state) (i: validpart) (fin: nat):
  LET m1= memory(conns(sys1)), m2= memory(conns(sys2)) IN
    initial_schultz(sys1) AND initial_schultz(sys2) AND
    length(m1) = length(m2) AND i < length(m1) AND
    psdequal(i,m1,m2) AND psdvalid(i,m1) AND psdvalid(i,m2) AND
    memoryrange(m1,base(i,m1),limit(i,m1)) =
      memoryrange(m2,base(i,m1),limit(i,m1))
    IMPLIES
      memoryrange(memory(conns(protected_system(sys1,0,fin))),
                  base(i,m1),limit(i,m1)) =
        memoryrange(memory(conns(protected_system(sys2,0,fin))),
                    base(i,m1),limit(i,m1))
```

Figure 8: Schultz invariant performance in PVS

6 Proving Schultz Invariant Performance

6.1 Background

There are two main lemmas needed to prove the invariant performance conjecture for Schultz. The "reset" lemma guarantees that resetting the AAMP-FV with Schultz leads to a reasonable state. The "maintenance" lemma guarantees that the code in Figure 5 maintains various invariants about the system that can be used to guarantee invariant performance. We briefly sketch the proof of the maintenance lemma.

The maintenance lemma is similar to the schultzip lemma in Figure 8, except rather than assume that the two Schultz states satisfy "initial Schultz" conditions they will be assumed to have "reasonable timer values" and be "safely executing the current user partition". Reasonable timer values means that the mucos timer is greater than the gong timer value plus the maximum user instruction time plus the maximum handler execution time (excepting the write of the mucos register). Safely executing the current user partition means that the PMU is initialized with the current base/limit pair, the processor is in user mode, and the executive PSD is initialized correctly.

The proof of the maintenance lemma is by induction on the time required to complete one user partition and partition swap. The base case is when the `fin` value occurs before the completion of the partition swap to the next partition, which we prove by showing that the only way to modify partition i's memory is if partition i is the current partition, in which case the memory is changed identically by the two executions. The inductive case is when the `fin` value occurs after the switch to the next partition. We show that the execution of a partition and the partition swap code maintains each of the hypotheses of this conjecture, which together with an inductive hypothesis suffices to prove the result.

Figure 6 shows a typical partition swap, where the gong timer generates an NMI which (after whatever latency is associated with finishing the current user instruction) transitions the machine into executive mode. After setting up

for the next partition the kernel then waits for the mucos timer timeout before starting the next partition. A second possibility exists for the timing of partition swapping, however: a partition can transition back to the kernel as the result of the execution of an illegal instruction, as suggested by Figure 7.

6.2 Maintenance Lemma Sublemmas

The maintenance lemma described in the previous section requires the proof of several sublemmas. Initial formalizations of these lemmas have been accomplished in PVS. Informal descriptions of these sublemmas are listed below.

1. Execution of the system can be decomposed into execution until the next transition to executive mode, and subsequent processing.
2. When transition to executive occurs the system is in a know state
3. When executing a user partition only the memory of that partition in changed.
4. When two executions of the partitioning system have an identical partition and have some identical executive values, the results of executing until the next executive transition are identical.
5. Partition swap code works to specification.
6. The specification of the partition swap code implies that the proper user partition will execute in user mode as a result of its execution.

6.3 A User Partition Memory Protection Proof

We have accomplished the proof of one of these lemmas for Schultz using PVS, that an executing user partitions' effect on memory is limited to its own memory space.[1]

This sublemma formalized in PVS is

```
partition_leaves_other_memory_untouched: LEMMA
  executing_partition(sys) IMPLIES
  visiblemem(val(pmubase(pmu(sys))),val(pmulimit(pmu(sys))), mem,
             memory(conns(protected_system(sys, cur, nextstop(sys, cur, fin))))) =
  visiblemem(val(pmubase(pmu(sys))),val(pmulimit(pmu(sys))), mem, memory(conns(sys)))
```

The function `visiblemem` takes four arguments - two addresses that describe a range of memory, and two memories, `memin` and `memout`. The result returned by this function is a new memory space with values that are identical to `memin` within the address range and identical to `memout` without.

[1]There are a total of 916 proofs accomplished in this proof, including all the subsidiary TCCs and sublemmas. Of these, 907 have proofs accepted by the PVS available in June 1999. The remaining lemmas that are left as unproved axioms primarily involve the equivalence of several views of bit-vector representations.

```
visiblemem(low,high,memin,memout): memory_space =
  (lambda (a: below(exp2(24))):
     if a>=low AND a<=high THEN memin(a) ELSE memout(a) ENDIF)
```

The function `protected_system` is the Schultz system, including processor and PMU, and appears as Figure 4. The PVS-checked proof of the lemma `partition_leaves_other_memory_untouched` assures us that no executing user task will write outside its memory region.

7 Summary

Statements of formal correctness that do not use an abstract execution model are rare. One example is the self-consistency checking work, wherein for example the operation of a processor's pipeline is specified using the same pipeline with "NOP"s inserted into the instruction stream [9]. Another is the symbolic simulation work whose objectives include regression testing of an evolving design by comparing symbolic execution of generations of designs [8].

The KIT correctness theorem, and all theorems that use an abstract model to specify system behavior, are not especially useful for justifying the integration of embedded applications. We believe that, from the standpoint of dependable embedded application integration, invariant performance is the right property to guarantee for partitioning systems.

References

[1] William Bevier. *A Verified Operating System Kernel.* PhD thesis, University of Texas at Austin, September 1987. Also available as ftp://ftp.cs.utexas.edu/pub/boyer/diss/bevier.ps.Z.

[2] William R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.

[3] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[4] William R. Bevier and William D. Young. Machine checked proofs of the design of a fault-tolerant circuit. *Formal Aspects of Computing*, 4:755–775, 1992.

[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[6] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[7] Colin Fidge, Peter Kearney, and Mark Utting. Formal specification and interactive proof of a simple real-time scheduler. Technical Report 94-11, Software Verification Research Centre, The University of Queensland, April 1994.

[8] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[9] Robert B. Jones, Carl-Johan H. Seger, and David L. Dill. Self-consistency checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[10] Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[11] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.

[12] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[13] J Strother Moore. *Piton – A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.

[14] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[15] David M. Russinoff. A mechanically checked proof of IEEE compliance of the AMD K5 floating-point square root microcode. Available as http://www.onr.com/user/russ/david/fsqrt.html, August 1996.

[16] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7 processor. Available at http://www.onr.com/user/russ/david/, January 28 1998.

[17] Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[18] Matthew Wilding. *Machine-Checked Real-Time System Verification*. PhD thesis, University of Texas at Austin, May 1996. Also available as ftp://ftp.cs.utexas.edu/pub/boyer/diss/wilding.ps.Z.

[19] Matthew Wilding, David Hardin, and David Greve. Invariant performance: A statement of task isolation useful for embedded application integration. *Proceedings of Dependable Computing for Critical Applications – DCCA-7*, 1999.

[20] Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356, 1997. (http://atb-www.larc.nasa.gov/Lfm97/).

Part IV

Methodologies and Tools to Automate Verification of Encapsulation

Overview

Effective specification and highly automated, formal verification of encapsulation properties have been major goals of the work performed for this contract; this focus has produced a wide range of reusable methods and significantly enhanced proof components for the PVS and SAL systems.

Rockwell Collins has refined the use of symbolic simulation by exploiting automated reasoning tools to derive behaviors from a formal system model, that is, effectively executing the model on a symbolic state. This approach retains much of the high assurance value of formal verification, while detouring the costs associated with formal proofs of correctness. The enclosed report [7] provides a brief overview of the methodology, while focusing on benchmarking to suggest PVS optimizations to enhance the system's ability to reason about code execution. A more detailed discussion of the methodology itself may be found in an account of work supported by Rockwell Collins on the design of the JEM1 microprocessor [3]. The key PVS optimization implemented to support symbolic simulation, namely the use of static analysis to provide safe destructive updates, is also described in an enclosed report [15].

SRI has developed several techniques to automate the formal verification of advanced microarchitectures, including the *Completion Functions Approach* [9], and the use of abstraction to reduce the complexity of proofs of partitioning properties [16]. SRI has also developed a systematic method for deductive verification of safety properties of concurrent programs, which works by strengthening a putative safety property into a disjunction of "configurations" that may be easily proven to be inductive [12].

SRI has focused on the development of tools and techniques to increase the scope, scale, automation, and utility of formal methods. Two enclosed papers [1,14] describe SRI's Symbolic Analysis Laboratory (SAL) and an instantiation of it that augments PVS with tools for abstraction, invariant generation, and program analysis (e.g., slicing). An extended abstract [10] suggests techniques for combining theorem proving, abstraction, and model checking in the SAL framework, and a final paper demonstrates the efficacy of a specific combination of theorem proving and model checking [13].

In E. A. Emerson and A. P. Sistla, editors, Computer-Aided Verification, CAV '2000, volume 1855 of Lecture Notes in Computer Science, pages 521-537, Chicago, IL, July 2000. ©Springer-Verlag

Verifying Advanced Microarchitectures that Support Speculation and Exceptions*

Ravi Hosabettu† Ganesh Gopalakrishnan† Mandayam Srivas‡

*The first and second authors were supported in part by NSF Grant No. CCR-9800928. The third author was supported in part by ARPA contract F30602-96-C-0204 and NASA contract NAS1-20334. The first author was also supported by a University of Utah Graduate Fellowship.

†Department of Computer Science, University of Utah, Salt Lake City, UT 84112

‡Computer Science Laboratory, SRI International, Menlo Park, CA 94025

Verifying Advanced Microarchitectures that Support Speculation and Exceptions *

Ravi Hosabettu[1], Ganesh Gopalakrishnan[1] and Mandayam Srivas[2]

[1] Department of Computer Science, University of Utah, Salt Lake City, UT 84112, hosabett,ganesh@cs.utah.edu

[2] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, srivas@csl.sri.com

Abstract. In this paper, we discuss the verification of a microprocessor involving a reorder buffer, a store buffer, speculative execution and exceptions at the microarchitectural level. We extend the earlier proposed *Completion Functions Approach* [HSG98] in a uniform manner to handle the verification of such microarchitectures. The key extension to our previous work was in systematically extending the abstraction map to accommodate the possibility of all the pending instructions being squashed. An interesting detail that arises in doing so is how the commutativity obligation for the program counter is proved despite the program counter being updated by both the instruction fetch stage (when a speculative branch may be entertained) and the retirement stage (when the speculation may be discovered to be incorrect). Another interesting detail pertains to how store buffers are handled. We highlight a new type of invariant in this work—one which keeps correspondence between store buffer pointers and reorder buffer pointers. All these results, taken together with the features handled using the completion functions approach in our earlier published work [HSG98,HSG99,HGS99], demonstrates that the approach is uniformly applicable to a wide variety of pipelined designs.

1 Introduction

Formal Verification of pipelined processor implementations against instruction set architecture (ISA) specifications is a problem of growing importance. A significant number of processors being sold today employ advanced features such as out-of-order execution, store buffers, exceptions that cause pending uncommitted instructions to be squashed, and speculative execution. Recently a number of different approaches [HSG99,McM98,PA98] have been used to verify simple out-of-order designs. To the best of our knowledge, no single formal verification technique has been shown to be capable of verifying processor designs that support

* The first and second authors were supported in part by NSF Grant No. CCR-9800928. The third author was supported in part by ARPA contract F30602-96-C-0204 and NASA contract NASI-20334. The first author was also supported by a University of Utah Graduate Fellowship.

all of these features and also apply to other processors such as those that perform out-of-order retirement. In this paper, we report our successful application of the *Completion Functions Approach* to verify an out-of-order execution design with a reorder buffer, a store buffer, exceptions and speculation, using the PVS [ORSvH95] theorem-prover, taking only a modest amount of time for the overall proof. This result, taken together with the earlier published applications of the completion functions approach [HSG98,HSG99,HGS99], demonstrates that the approach is uniformly applicable to a wide variety of pipelined designs.

One of the challenges posed in verifying a combination of the above mentioned advanced features is that the resulting complex interaction between data and control usually overwhelms most automatic methods, whether based on model checking or decision procedures. One of the main contributions of this work is that we develop a way of cleanly decomposing the squashing of instructions from normal execution. These decomposition ideas are applicable to theorem proving or model checking or combined methods.

Our basic approach is one of showing that any program run on the specification and the implementation machines returns identical results. This verification is, in turn, achieved by identifying an abstraction map ABS that relates implementation states to corresponding specification states. The key to make the above technique work efficiently in practice is a proper definition of ABS. As we showed, in our earlier work [HSG98], one should ideally choose an approach to constructing ABS that is not only simple and natural to carry out, but also derives other advantages, the main ones being *modular verification* that helps localize errors, and *verification reuse* that allows lemmas proved about certain pipeline stages to be used as rewrite rules in proving other stages. In [HSG98], we introduced such a technique to define ABS called the *Completion Functions Approach*. In subsequent work [HSG99,HGS99,Hos99], we demonstrated that the completion functions approach can be applied uniformly to a wide variety of examples that include various advanced pipelining features. An open question in our previous work was whether combining out-of-order execution with exceptions and speculation would make the task of defining completion functions cumbersome and the approach impractical.

In this paper, we demonstrate that the completion functions approach is robust enough to be used effectively for such processors, that is, (i) the specification of completion functions are still natural, amounting to expressing knowledge that the designer already has; (ii) verification proceeds incrementally, facilitating debugging and error localization; (iii) mistakes made in specifying completion functions never lead to false positives; and (iv) verification conditions and most of the supporting lemmas needed to finish a proof can be generated systematically, if not automatically. They can also be discharged with a high degree of automation using strategies based on decision procedures and rewriting. These observations are supported by our final result: a processor design supporting superscalar execution, store buffers, exceptions, speculative branch prediction, and user and supervisor modes could be fully verified in 265 person hours. This,

we believe, is a modest investment in return for the significant benefits of design verification.

Some of the highlights of the work we report are as follows. Given that our correctness criterion is one of showing a commutativity obligation between implementation states and specification states, the abstraction map used in the process must somehow accommodate the possibility of instructions being squashed. We show how this is accomplished. This leads us to a verification condition with two parts, one pertaining to the processor states being related before and after an implementation transition, and the other relating to the squashing predicate itself. Next, we show how the commutativity obligation for the program counter is obtained despite the program counter being updated by both the instruction fetch stage (when a speculative branch may be entertained) and the retirement stage (when the speculation may be discovered to be incorrect). We also show how the store buffer is handled in our proof. We detail a new type of invariant in this work, which was not needed in our earlier works. This invariant keeps correspondence between store buffer pointers and reorder buffer pointers.

2 Processor Model

At the specification level, the state of the processor is represented by a register file, a special register file accessed only by privileged/special instructions, a data memory, a mode flag, a program counter and an instruction memory. The processor operating mode (one of user/supervisory) is maintained in the mode flag. User mode instructions are an `alu` instruction for performing arithmetic and logical operations, `load` and `store` instructions for accessing the data memory, and a `beq` instruction for performing conditional branches. Three additional privileged instructions are allowed in the supervisory mode: `rfeh` instruction for returning from an exception handler, and `mfsr` and `mtsr` instructions for moving data from and to the special register file. Three types of exceptions are possible: arithmetic exception raised by an `alu` instruction, data access exception raised by `load` and `store` instructions when the memory address is outside legal bounds (two special registers maintain the legal bounds, and this is checked only in user mode), and an illegal instruction exception. When an exception is raised, the processor saves the address of the faulting instruction in a special register and jumps to an exception handler assuming supervisory mode in the process. After processing a raised exception, the processor returns to user mode via the `rfeh` instruction.

An implementation model of this processor is shown in Figure 1. A reorder buffer, implemented as a circular FIFO queue with its tail pointing to the earliest issued instruction and head pointing to the first free location in the buffer, is used to maintain program order, to permit instructions to be committed in that order. Register translation tables (regular and special) provide the identity of the latest pending instruction writing a particular register. "Alu/Branch/Special Instr. Unit" (referred to as ABS Unit) executes `alu`, `beq` and all the special instructions. The reservation stations hold the instructions sent to this unit

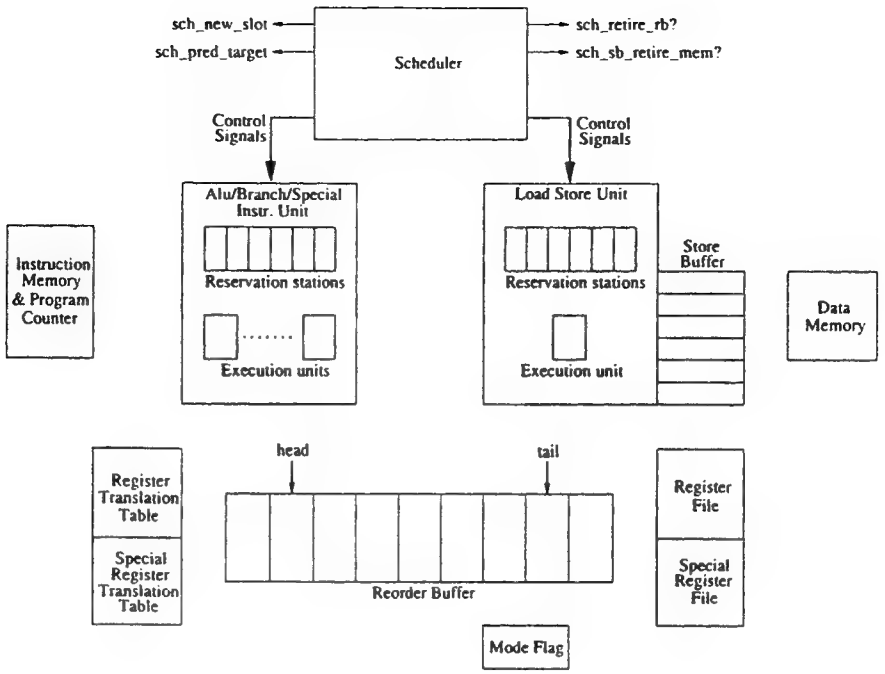


Fig. 1. The block diagram model of our implementation

until they are ready to be dispatched onto an appropriate execution unit. These instructions are executed out of program order by the multiple execution units present in the ABS Unit. Instructions `load` and `store` are issued to the "Load Store Unit" (referred to as LS Unit) where the reservation stations form a circular FIFO queue storing the instructions in their program order. (Again, tail points to the earliest instruction and head points to the first free reservation station.) These instructions are executed in their program order by the single execution unit present in the LS Unit. For a `store` instruction, the memory address and the value to be stored are recorded in an entry in the *store buffer*, and the value is later written into the data memory. The store buffer is again implemented as a circular FIFO queue, with head and tail pointers, keeping the instructions to be written to the data memory in their program order. When two store buffer entries refer to the same memory address, the latest one has a flag set. A `load` instruction first attempts an associative search in the store buffer using the memory address. If multiple store buffer entries have the same address, the search returns the value of the latest entry. If the search does not find a matching entry, the data for that address is returned from the data memory. A scheduler controls the movement of the instructions through the execution pipeline (such as being dispatched, executed etc.) and its behavior is modeled by axioms (to allow us to concentrate on the processor "core"). Instructions are fetched from

the instruction memory using a program counter; and the implementation also takes a no_op input, which suppresses an instruction fetch when asserted.

An instruction is *issued* by allocating an entry for it at the head of the reorder buffer and (depending on the instruction type) either a free reservation station (`sch_new_slot`) in the ABS Unit or a free reservation station at the head of the queue of reservation stations in the LS Unit. If the instruction being issued is a branch instruction, then the program counter is modified according to a *predicted* branch target address (`sch_pred_target`, an unconstrained arbitrary value), and in the next cycle the new instruction is fetched from this address. No instruction is issued if there are no free reservation stations/reorder buffer entries or if no_op is asserted or if the processor is being restarted (for reasons detailed later). The RTT entry corresponding to the destination of the instruction is updated to reflect the fact that the instruction being issued is the latest one to write that register. If the source operands are not being written by previously issued pending instructions (checked using the RTT) then their values are obtained from the register file, otherwise the reorder buffer indices of the instructions providing the source operands are maintained (in the reservation station). Issued instructions wait for their source operands to become ready, monitoring all the execution units if they produce the values they are waiting for. An instruction can be *dispatched* when its source operands are ready and a free execution unit is available [1]. In case of the LS Unit, only the instruction at the tail of the queue of reservation stations is dispatched. As soon as an instruction is dispatched, its reservation station is freed. The dispatched instructions are *executed* and the results are *written back* to their respective reorder buffer entries as well as forwarded to those instructions waiting for this result. If an exception is raised by any of the executing instructions, then a flag is set in the reorder buffer entry to indicate that fact. In case of a `store` instruction, the memory address and the value to be stored are written into a store buffer entry instead of the reorder buffer entry when the `store` instruction does not raise an exception (other information such as the "ready" status etc. are all written into the reorder buffer entry). The control signals from the scheduler determine the timings of this movement of the instructions in the execution pipeline.

The instruction at the tail of the reorder buffer is committed to the architecturally visible components, when it is done executing (at a time determined by `sch_retire_rb?`). If it is a `store` instruction, then the corresponding store buffer entry is marked *committed* and later written into the data memory (at a time determined by `sch_sb_retire_mem?`). Also, if the RTT entry for the destination of the instruction being retired is pointing to the tail of the reorder buffer, then that RTT entry is updated to reflect the fact that the value of that register is in the appropriate register file. If the instruction at the tail of the reorder buffer has raised an exception or if it is a mis-predicted branch or if it is a `rfeh` in-

[1] Multiple instructions can be simultaneously dispatched, executed and written back in one clock cycle. However, for simplicity, we do not allow multiple instruction issue or retirement in a single clock cycle.

struction, then the rest of the instructions in the reorder buffer are squashed and the processor is restarted by resetting all of its internal (non-observable) state.

3 The Completion Functions Approach

The key idea in proving the correctness of pipelined microprocessors is to discover a formal correspondence between the execution of the implementation and the specification machines. The completion functions approach suggests a way of constructing this abstraction in a manner that leads to an elegant decomposition of the proof. In the first subsection, we briefly discuss the correctness criterion we use. In the second subsection, we describe the different steps in constructing a suitable abstraction function for the example under consideration. In the third subsection, we discuss how to decompose the proof into verification conditions, the proof strategies used in discharging these obligations, and the invariants needed in our approach. The PVS specifications and the proofs can be found at [Hos99].

3.1 Correctness Criterion

We assume that the pipelined implementation and the ISA-level specification are provided in the form of transition functions, denoted by `I_step` and `A_step` respectively. The specification machine state is made up of certain components chosen from the implementation machine called the observables. The function `projection` extracts these observables given an implementation machine state. The state where the pipelined machine has no partially executed instructions is called a *flushed* state.

We regard a pipelined processor implementation to be *correct* if the behavior of the processor starting in a flushed state, executing a program, and terminating in a flushed state is matched by the ISA level specification machine whose starting and terminating states are in direct correspondence with those of the implementation processor through `projection`. This criterion is shown in Figure 2(a) where `n` is the number of implementation machine transitions in a run of the pipelined machine and `m` corresponds to the number of instructions executed in the specification machine by this run. An additional correctness criterion is to show that the implementation machine is able to execute programs of all lengths, that is, it does not get into a state where it refuses to accept any more new instructions. In this paper, we concentrate on proving the correctness criterion expressed in Figure 2(a) only.

The criterion shown in Figure 2(a) spanning an entire sequential execution can be established with the help of induction once a more basic *commutativity obligation* shown in Figure 2(b) is established on a single implementation machine transition. This criterion states that if the implementation machine starts in an arbitrary state `q` and the specification machine starts in a corresponding specification state (given by an abstraction function `ABS`), then after executing a transition their new states correspond. `A_step_new` stands for zero or more

applications of `A_step`. The number of instructions executed by the specification machine corresponding to an implementation transition is given by a user defined *synchronization* function. Our method further verifies that the ABS function chosen corresponds to `projection` on flushed states, that is, `ABS(fs) = projection(fs)` holds on flushed states, thus helping debug ABS. The user may also need to discover invariants to restrict the set of implementation states considered in the proof of the commutativity obligation and prove that it is closed under `I_step`.

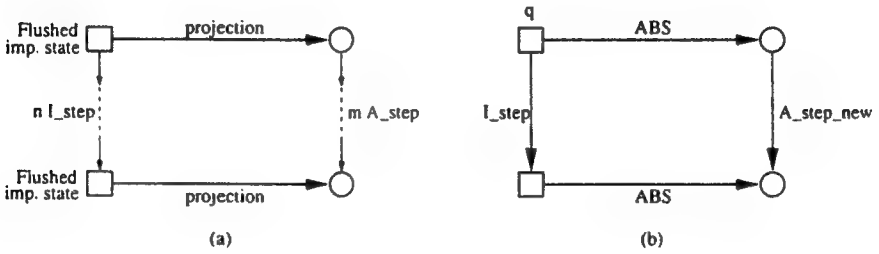


Fig. 2. Pipelined microprocessor correctness criterion

The crux of the problem here is to define a suitable abstraction function relating an implementation state to a specification state. The completion functions approach suggests a way of doing this in a manner that leads to an elegant decomposition of the proof. We now detail how this is achieved for our example processor.

3.2 Compositional construction of the abstraction function

The first step in defining the abstraction function is to identify all the unfinished instructions in the processor and their program order. In this implementation, the processor (when working correctly) stores all the currently executing instructions in their program order in the reorder buffer. We identify an instruction in the processor with its reorder buffer index, that is, we refer to instruction at reorder buffer index `rbi` as just instruction `rbi`[2]. In addition to these, the store buffer has certain committed `store` instructions yet to be written into the data memory, recorded in their program order. These `store` instructions are not associated with any reorder buffer entry and occur earlier in the program order than all the instructions in the reorder buffer.

[2] Brief explanation of some of the notation used throughout rest of the paper: `q` refers to an arbitrary implementation state, `s` the scheduler output, `i` the processor input, `I_step(q,s,i)` the next state after an implementation transition. We sometimes refer to predicates and functions defined without explicitly mentioning their arguments, when this causes no confusion.

Having determined the program order of the unfinished instructions, the second step is to define a completion function for every unfinished instruction in the pipeline. Each completion function specifies the *desired effect* on the observables of completing a particular unfinished instruction assuming those that are ahead of it (in the program order) are completed. The completion functions, which map an implementation state to an implementation state, leave all non-observable state components unchanged. However not every instruction in the pipeline gets executed completely and updates the observables. If an instruction raises an exception or if the target address is mis-predicted for a branch instruction, then the instructions following it must be squashed. To specify this behavior, we define a squashing predicate for every unfinished instruction that is true exactly when the unfinished instruction can cause the subsequent instructions (in the program order) to be squashed. The completion function for a given instruction updates the observables only if the instruction is not squashed by any of the instructions preceding it.

We now elaborate on specifying the completion functions and the squashing predicates for the example under consideration. An unfinished instruction `rbi` in the processor can be in one of the following seven phases of execution: Issued to ABS Unit or to LS Unit (*issued_abs* or *issued_lsu*), dispatched in either of these units (*dispatched_abs* or *dispatched_lsu*), executed in either of these units (*executed_abs* or *executed_lsu*) or written back to the reorder buffer (*writtenback*). A given unfinished instruction is in one of these phases at any given time and the information about this instruction (the source values, destination register etc) is held in the various implementation components. For each instruction phase "ph", we define a predicate "Instr_ph?" that is true when a given instruction is in phase "ph", a function "Action_ph" that specifies what ought to be the effect of completing an instruction in that phase, and a predicate "Squash_rest?_ph" that specifies the conditions under which an instruction in that phase can squash all the subsequent instructions. We then define a single parameterized completion function and squashing predicate (applicable to all the unfinished instructions in the reorder buffer) as shown in 1. We similarly define (a parameterized) completion function for the committed `store` instructions in the store buffer. These `store` instructions can only be in a single phase, that is, *committed*, and they do not cause the subsequent instructions to be squashed. (A `store` instruction that raises an exception is not entered into the store buffer.)

1

```
% state_I: impl. state type. rbindex: reorder buffer index type.
Complete_instr(q:state_I, rbi:rbindex, kill?:bool): state_I =
  IF kill? THEN q
  ELSIF Instr_writtenback?(q,rbi) THEN Action_writtenback(q,rbi)
  ELSIF Instr_executed_lsu?(q,rbi) THEN Action_executed_lsu(q,rbi)
  ELSIF ... Similarly for other phases ... ENDIF

Squash_rest?_instr(q:state_I, rbi:rbindex): bool =
  IF Instr_writtenback?(q,rbi) THEN Squash_rest?_writtenback(q,rbi)
  ELSIF Instr_executed_lsu?(q,rbi) THEN Squash_rest?_executed_lsu(q,rbi)
  ELSIF ... Similarly for other phases ... ENDIF
```

In this implementation, when an instruction is in the *writtenback* phase, its reorder buffer entry has the result value and destination of the instruction, and also enough information to determine whether it has raised any exceptions or has turned out to be a mis-predicted branch. `Action_writtenback` and `Squash_rest?_writtenback` are then defined using this information about the instruction. Similarly, we define the "Action"s and the "Squash_rest?"s for the other phases. When an instruction is in an execution phase where it has not yet read its operands, the completion function obtains the operands by simply reading them from the observables. The justification is that the completion functions are composed in their program order in constructing the abstraction function (described below), and so we talk of completing a given instruction in a context where the instructions ahead of it are completed.

```
% Complete_Squash_rest?_till returns a tuple.                          2
% proj_1 and proj_2 extracts the first and the second components.
% rbindex_p is type ranging from 0 to the size of the reorder buffer.
Complete_Squash_rest?_till(q:state_I,rbi_ms:rbindex_p):
                                              RECURSIVE [state_I,bool] =
 IF rbi_ms = 0 THEN (q,FALSE)
 ELSE LET t = Complete_Squash_rest?_till(q,rbi_ms-1),
          x = proj_1(t), y = proj_2(t) IN
    (Complete_instr(x,measure_fn_rbi(q,rbi_ms),y), %% 1st component.
     Squash_rest?_instr(x,measure_fn_rbi(q,rbi_ms)) OR y) %% 2nd one.
 ENDIF
 MEASURE rbi_ms

Complete_till(q:state_I,rbi_ms:rbindex_p): state_I =
 proj_1(Complete_Squash_rest?_till(Complete_committed_in_sb_till(
                                   q,lsu_sb_commit_count(q)),rbi_ms))

Squash_rest?_till(q:state_I,rbi_ms:rbindex_p): bool =
 proj_2(Complete_Squash_rest?_till(Complete_committed_in_sb_till(
                                   q,lsu_sb_commit_count(q)),rbi_ms))

% state_A is the specification state type.
ABS(q: state_I): state_A = projection(Complete_till(q,rb_count(q)))
```

The final step is to construct the abstraction function (that has the cumulative effect of flushing the pipeline) by completing all the unfinished instructions in their program order. A given instruction is to be killed, that is, the `kill?` argument of `Complete_instr` is true, when the squashing predicate is true for any of the instructions ahead of that given instruction. In order to define an ordering among the instructions, we define a measure function `rbi_measure_fn` that associates a measure with every instruction in the reorder buffer such that the tail has measure one and successive instructions have a measure one greater than the previous instruction. So the instructions with lower measures occur earlier in the program order than instructions with higher measures. The function `measure_fn_rbi` returns the reorder buffer index of the instruction with the given measure. To define the abstraction function, we first define a recursive function

Complete_Squash_rest?_till that completes the instructions and computes the disjunction of the squashing predicates from the tail of the reorder buffer till a given unfinished instruction, as shown in [2]. Complete_committed_in_sb_till is a similar recursive function that completes all the committed store instructions in the store buffer. We can then define the abstraction function by first completing all the committed instructions in the store buffer (they are earlier in the program order that any instruction in the reorder buffer) and then completing all the instructions in the reorder buffer. So we define Complete_till and Squash_rest?_till as shown in [2], and then in constructing the abstraction function ABS, we instantiate the Complete_till definition with the measure of the latest instruction in the reorder buffer. The implementation variable rb_count maintains the number of instructions in the reorder buffer, and hence corresponds to the measure of the latest instruction.

3.3 Decomposing the proof

The proof of the commutativity obligation is split into different cases based on the structure of the synchronization function. In this example, the synchronization function returns zero when the processor is restarted or if the squashing predicate evaluates to true for any of the instructions in the reorder buffer (i.e., Squash_rest?_till(q,rb_count(q)) is true) or if no new instruction is issued. Otherwise it returns one, and we consider each of these cases separately. We discuss proving the commutativity obligation for register file rf and program counter pc only. The proofs for the special register file, mode flag and data memory are similar to that rf, though in the case of data memory, one needs to take into account the additional details regarding the committed instructions in the store buffer. The proof for instruction memory is straight-forward as it does not change at all.

We first consider an easy case in the proof of the commutativity obligation (for rf), that is, when the processor is being restarted in the current cycle (restart_proc is true).

- The processor discards all the executing instructions in the reorder buffer, and sets rb_count and lsu_sb_commit_count to zeros. So Complete_till will be vacuous on the implementation side of the commutativity obligation (the side on which I_step(q,s,i) occurs), and the expression on the implementation side simplifies to rf(I_step(q,s,i)).
- Whenever the processor is being restarted, the instruction at the tail of the reorder buffer is causing the rest of the instructions to be squashed, so Squash_rest?_till(q,1) ought to be true. (Recall that the tail of the reorder buffer has measure one.) We prove this, and then from the definition of Complete_Squash_rest?_till in [2], it follows that the kill? argument is true for all the remaining instructions in the reorder buffer, and hence these do not affect rf. Also, the synchronization function returns zero when the processor is being restarted. So the expression on the specification side of the commutativity obligation simplifies to rf(Complete_till(q,1)).

– We show that the rf(I_step(q,s,i)) and rf(Complete_till(q,1)) are indeed equal by expanding the definitions occurring in them and simplifying.

Now assume that restart_proc is false. We first postulate certain verification conditions, prove them, and then use them in proving the commutativity obligation. Consider an arbitrary instruction rbi. Though the processor executes the instructions in an out-of-order manner, it commits the instructions to the observables only in their program order. This suggests that the effect on rf of completing all the instructions till rbi is the same in the states q and I_step(q,s,i). Similarly, the truth value of the disjunction of the squashing predicates till rbi is the same in the states q and I_step(q,s,i). This verification condition Complete_Squash_rest?_till_VC is shown in 3. This is proved by an induction on rbi_ms[3] (the measure corresponding to instruction rbi).

3

```
% valid_rb_entry? predicate tests whether rbi is within the
% reorder buffer bounds.
Complete_Squash_rest?_till_VC: LEMMA
FORALL(rbi_ms:rbindex): LET rbi = measure_fn_rbi(q,rbi_ms) IN
((valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s,i)) IMPLIES (
rf(Complete_till(q,rbi_ms)) = rf(Complete_till(I_step(q,s,i),rbi_ms)) AND
Squash_rest?_till(q,rbi_ms) = Squash_rest?_till(I_step(q,s,i),rbi_ms)))
```

As in the earlier proofs based on completion functions approach [HSG99,HSG98], we decompose the proof of Complete_Squash_rest?_till_VC into different cases based on how an instruction makes a transition from its present phase to its next phase. Figure 3 shows the phase transitions for an instruction rbi in the reorder buffer (when the processor is not restarted) where the predicates labeling the arcs define the conditions under which those transitions take place. The Figure also shows the three transitions for a new instruction entering the processor pipeline. Having identified these predicates, we prove that those transitions indeed take place in the implementation machine. For example, we prove that an instruction rbi in phase *dispatched_lsu* (D_lsu in the Figure) goes to *executed_lsu* phase in I_step(q,s,i) if Execute_lsu? predicate is true, otherwise it remains in *dispatched_lsu* phase.

We now return to the proof of Complete_Squash_rest?_till_VC and consider the induction argument (i.e., rbi_ms is not equal to 1). The proof outline is as follows:

– Expand the Complete_till and the Squash_rest?_till definitions on both sides of Complete_Squash_rest?_till_VC and unroll the recursive definition of Complete_Squash_rest?_till once.
– Consider the first conjunct (i.e., one corresponding to rf). The kill? argument to Complete_instr is Squash_rest?_till(q,rbi_ms-1) on the left

[3] Since the measure function is dependent on the tail of the reorder buffer, and since the tail can change during an implementation transition, the measure needs to be adjusted on the right hand side of Complete_Squash_rest?_till_VC to refer to the same instruction. This is a detail which we ignore in this paper for the ease of explanation, and use just rbi_ms.

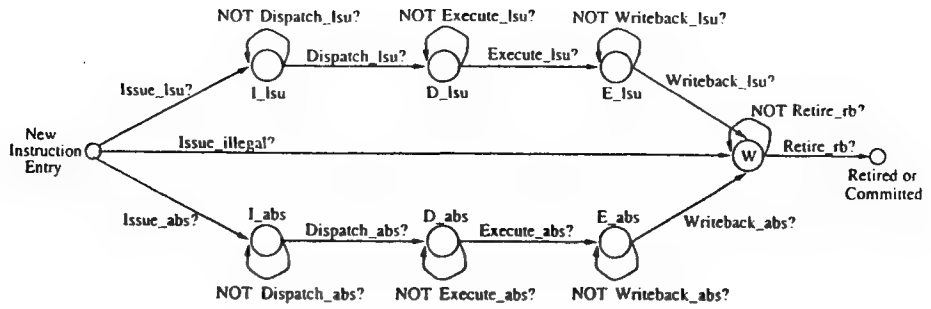


Fig. 3. The various phases an instruction can be in and transitions between them (when the processor is not being restarted). Also, the three transitions for an instruction entering the processor are shown.

hand side and `Squash_rest?_till(I_step(q,s,i),rbi_ms-1)` on the right hand side, and these have the same truth value by the induction hypothesis. When it is true, the left hand side reduces to `rf(Complete_till(q,rbi_ms-1))` and the right hand side to `rf(Complete_till(I_step(q,s,i),rbi_ms-1))` which are equal by the induction hypothesis. When it is false, the proof proceeds as in our earlier work [HSG99]. We consider the possible phases `rbi` can be in and whether or not, it makes a transition to its next phase. Assume `rbi` is in *dispatched_abs* phase and the predicate `Execute_abs?` is true. Then, in `I_step(q,s,i)`, `rbi` is in *executed_abs* phase. By the definition of `Complete_instr`, the left hand side of the verification condition simplifies to `rf(Action_dispatched_abs(Complete_till(q,rbi_ms-1),rbi_ms))` and the right hand side reduces to `rf(Action_executed_abs(Complete_till(I_step(q,s,i),rbi_ms-1),rbi_ms))`. The proof now proceeds by expanding these "Action" function definitions, using the necessary invariant properties and simplifying. The induction hypothesis will be used to infer that the register file contents in the two states `Complete_till(q,rbi_ms-1)` and `Complete_till(I_step(q,s,i),rbi_ms-1)` are equal, as those two terms appear when the "Action" definitions are expanded. Overall, the proof decomposes into 14 cases for the seven phases `rbi` can be in.

- Consider the second conjunct of `Complete_Squash_rest?_till_VC`. Using the induction hypothesis, this reduces to showing that the two predicates `Squash_rest?_instr(Complete_till(q,rbi_ms-1),rbi_ms)` and `Squash_rest?_instr(Complete_till(I_step(q,s,i),rbi_ms-1),rbi_ms)` have the same truth value. This proof again proceeds as before by a case analysis on the possible phases `rbi` can be in and whether or not, it makes a transition to its next phase. The proof again decomposes into 14 cases for the seven phases `rbi` can be in.

For the program counter, however, it is not possible to relate its value in states `q` and `I_step(q,s,i)` by considering the effect of instructions *one at a*

time in their program order as was done for `rf`. This is because `I_step` updates `pc` if a new instruction is fetched, either by incrementing it or by updating it according to the speculated branch target address, but this new instruction is the latest one in the program order. However, if the squashing predicate is true for any of the executing instructions in the reorder buffer, then completing that instruction modifies the `pc` with a higher precedence, and the `pc` ought to be modified in the same way in both `q` and `I_step(q,s,i)`. This observation suggests a verification condition on `pc`, shown in 4. This verification condition is again proved by an induction on `rbi_ms`, and its proof is decomposed into 14 cases based on the instruction phase transitions as in the earlier proofs.

```
pc_remains_same_VC: LEMMA                                              4
FORALL(rbi_ms:rbindex): LET rbi = measure_fn_rbi(q,rbi_ms) IN
 (valid_rb_entry?(q,rbi) AND NOT restart_proc?(q,s,i) AND
       Squash_rest?_till(q,rbi_ms)) IMPLIES
 pc(Complete_till(q,rbi_ms)) = pc(Complete_till(I_step(q,s,i),rbi_ms))
```

Now we come to the proof of the commutativity obligation, where we use the above lemmas after instantiating them with `rb_count`. We consider the different remaining cases in the definition of the synchronization function in order—`Squash_rest?_till(q,rb_count(q))` is true, no new instruction is issued or the three transitions corresponding to a new instruction being issued as shown in Figure 3.

- When `Squash_rest?_till(q,rb_count(q))` is true, the `kill?` argument for the new instruction fetched (if any) will be true in `I_step(q,s,i)` since `Squash_rest?_till` has the same truth value in states `q` and `I_step(q,s,i)`. Hence on the implementation side of the commutativity obligation, there is no new instruction executed. On the specification side, the synchronization function returns zero, so `A_step_new` is vacuous. The proof can then be accomplished using `Complete_Squash_rest?_till_VC` (for the register file) and `pc_remains_same_VC` (for the program counter).
- The proof when no new instruction is issued or when one is issued is similar to the proof in our earlier work [HSG99]. For example, if the issued instruction is in *issued_lsu* phase in `I_step(q,s,i)`, then we have to prove that completing this instruction according to `Action_issued_lsu` has the same effect on the observables as executing a specification machine transition.

Correctness of the feedback logic: Whenever there are data dependencies among the executing instructions, the implementation keeps track of them and forwards the results of the execution to all the waiting instructions. The correctness of this feedback logic, both for the register file and the data memory, is expressed in a similar form as in our earlier work [HSG99]. For example, a `load` instruction obtains the value from the store buffer if there is an entry with the matching address (using associative search), otherwise it reads the value from the data memory. Consider the value obtained when all the instructions ahead of the `load` instruction are completed, and then the data memory is read. This value and the value returned by the feedback logic ought to be equal. The verification condition for the correctness of the feedback logic for data memory is

based on the above observation. It will be used in the proof of the commutativity obligation and the proof of this verification condition itself is based on certain invariants.

Invariants needed: Many of the invariants needed like the exclusiveness and the exhaustiveness of instruction phases, and the invariants on the feedback logic for the register file and data memory are similar to the ones needed in our earlier work [HSG99]. We describe below one invariant that was not needed in the earlier work.

The LS Unit executes the `load` and `store` instructions in their program order. These instructions are stored in their program order in the reservation stations in the LS Unit and in the store buffer. It was necessary to use these facts during the proof and it was expressed as follows (for the reservation stations in LS Unit): Let `rsi1` and `rsi2` be two instructions in the reservation stations in the LS Unit. `rsi1_ptr` and `rsi2_ptr` point to the reorder buffer entries corresponding to these instructions respectively. Let `lsu_rsi_measure_fn` be a measure function defined on the LS Unit reservation station queue similar to `rbi_measure_fn`. If `rsi1` has a lower/higher measure than `rsi2` according to `lsu_rsi_measure_fn`, then `rsi1_ptr` has a lower/higher measure than `rsi2_ptr` according to a `rbi_measure_fn`.

PVS proof effort organization: This exercise was carried out in four phases. In the first phase, we "extrapolated" certain invariants and properties from the earlier work, and this took 27 person hours. In the second phase, we formulated and proved the invariants and certain other properties on the store buffer, and this took 54 person hours. In the third phase, we formulated and proved all the verification conditions about the observables and the commutativity obligation, and this took 131 person hours. In the fourth phase, we proved the necessary invariants about the feedback logic and its correctness, and this took 53 person hours. So the entire proof was accomplished in 265 person hours.

Related work: There is one other reported work on formally verifying the correctness of a pipelined processor of comparable complexity. In [SH99], Sawada and Hunt construct an explicit intermediate abstraction in the form of a table called MAETT, express invariant properties on this and prove the final correctness from these invariants. They report taking 15 person months. Also, their approach is applicable to fixed size instantiations of the design only. Various other approaches have been proposed to verify out-of-order execution processors recently [McM98,PA98,JSD98,BBCZ98,CLMK99,BGV99], but none of these have been so far demonstrated on examples with a similar set of features as we have handled.

Example verified	Effort spent doing the proof
EX1.1 and EX1.2	2 person months
EX3.1	13 person days
EX2.1	19 person days
EX3.2	7 person days
EX2.2	34 person days

Table 1. Examples verified and the effort needed.

4 Experimental Results and Concluding Remarks

We have applied our methodology to verify six example processors exhibiting a wide variety of implementation issues, and implemented our methodology in PVS [ORSvH95]. Our results to date are summarized in Table 1. This table summarizes the manual effort spent on each of the examples, listing them in the order we verified them. The first entry includes the time to learn PVS[4]. Each verification effort built on the earlier efforts, and reused some the ideas and the proof machinery.

The processor described in this paper is listed as EX2.2. In contrast, EX1.1 is a five stage pipeline implementing a subset of the DLX architecture, EX1.2 is a dual-issue version of the same architecture, and EX2.1 is a processor with a reorder buffer and only arithmetic instructions. We also considered two examples that allowed out-of-order completion of instructions: EX3.1 allowed certain arithmetic instructions to bypass certain other arithmetic instructions when their destinations were different, and EX3.2 implemented Tomasulo's algorithm without a reorder buffer and with arithmetic instructions only.

In conclusion, the completion functions approach can be used effectively to verify a wide range of processors against their ISA-level specifications. We have articulated a systematic procedure by which a designer can formulate a very intuitive set of completion functions that help define the abstraction function, and then showed how such a construction of the abstraction function leads to decomposition of the proof of the commutativity obligation. We have also presented how the designer can systematically address details such as exceptions and feedback logic. Design iterations are also greatly facilitated by the completion functions approach due to the incremental nature of the verification, as changes to a pipeline stage do not cause ripple-effects of changes across the whole specification; global re-verification can be avoided because of the layered nature of the verification conditions. Our future work will be directed at overcoming the current limitations of the completion functions approach, by seeking ways to automate invariant discovery, especially pertaining to the control logic of processors.

[4] By the first author who did all the verification work.

References

[BBCZ98] Sergey Berezin, Armin Biere, Edmund Clarke, and Yunshan Zu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In Gopalakrishnan and Windley [GW98], pages 369–386.

[BGV99] Randal Bryant, Steven German, and Miroslav Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In Halbwachs and Peled [HP99], pages 470–482.

[CLMK99] Byron Cook, John Launchbury, John Matthews, and Dick Kieburtz. Formal verification of explicitly parallel microprocessors. In Pierre and Kropf [PK99], pages 23–36.

[GW98] Ganesh Gopalakrishnan and Phillip Windley, editors. *Formal Methods in Computer-Aided Design, FMCAD '98*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, USA, November 1998. Springer-Verlag.

[HGS99] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In Pierre and Kropf [PK99], pages 8–22.

[Hos99] Ravi Hosabettu. The Completion Functions Approach homepage, 1999. At address http://www.cs.utah.edu/~hosabett/cfa.html.

[HP99] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[HSG98] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In Hu and Vardi [HV98], pages 122–134.

[HSG99] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Halbwachs and Peled [HP99], pages 47–59.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.

[JSD98] Robert Jones, Jens Skakkebaek, and David Dill. Reducing manual abstraction in formal verification of out-of-order execution. In Gopalakrishnan and Windley [GW98], pages 2–17.

[McM98] Ken McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Hu and Vardi [HV98], pages 110–121.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PA98] Amir Pnueli and Tamarah Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In Gopalakrishnan and Windley [GW98], pages 351–368.

[PK99] Laurence Pierre and Thomas Kropf, editors. *Correct Hardware Design and Verification Method, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1999. Springer-Verlag.

[SH99] J. Sawada and W.A. Hunt, Jr. Results of the verification of a comples pipelined machine model. In Pierre and Kropf [PK99], pages 313–316.

Automating Partition Proofs

Mandayam K. Srivas
Computer Science Laboratory
SRI International
Menlo Park, CA 94025 USA

Abstract

We study and contrast the role played by abstraction, invariant generation, and automation in deductive and algorithmic verification using a small, but realistic embedded system.

1 Introduction

Key techniques that determine the successful application of formal verification to real systems, whether the method of proof is deductive or algorithmic, include abstraction, invariant generation, automation, and scaling. In this paper, we study and contrast the role played by these techniques by verifying a small but significant example using a deductive method based on rewriting and decision procedures, and an algorithmic method based on symbolic model checking.

The example illustrates formal analysis of a particular kind of safety property, namely a *partitioning* property about a hardware/software system architecture that implements a safe partitioning mechanism. Partitioning [5] is an important problem in Integrated Modular Avionics (IMA), where software that supports a collection of avionics tasks is implemented using a common, shared computing platform. This shared architecture provides pathways for undesired interference between the different tasks supported by the software. A partitioning mechanism in an IMA architecture is intended to prevent design faults in one task from interfering with the behavior of other tasks. The advantage of having a partitioning mechanism is that not every task has to be assured to the same level regardless of its criticality. As long as the partitioning mechanism and the critical tasks are subjected to a high level of assurance, it is not necessary to worry about the faults in noncritical tasks affecting critical ones, even if the noncritical tasks are not formally verified.

The partitioning architecture we analyze is built using a microcoded processor, the AAMP-FV, whose microcode has been partially verified [8], a Partition Management Unit (PMU), and an unprotected MEMORY. A piece of kernel code implements the partitioning mechanism in hardware. A partitioning property may be expressed as a safety property on execution traces. Our formal model is written as a program interpreter for AAMP-FV code with specially defined functions to model the hardware mechanisms. The system is centrally controlled and deterministic.

This architecture is a good candidate for proof by rewriting and decision procedures because it uses a data-dominated design. Given a sufficiently strong invariant on the set of reachable states, the proof can, in principle, be automated in PVS using rewriting and decision procedures. Rockwell Collins attempted a partial verification of the design, using a proof based on symbolic simulation of the actual kernel code. Our goal in re-verifying the design is to explore the use of appropriate abstractions to reduce the complexity of the proof.

We employ two abstractions in the deductive proof. The first abstracts the kernel code into a set of constraints, referred to as `kernel_code_invariants`, that must be satisfied by every transition made by execution of the code. The second abstracts

the level in the definition hierarchy of the transition function to which the transition function is symbolically simulated. Both abstractions are constructed with respect to the property to be verified. Predictably, the most challenging part of the proof using symbolic simulation and decision procedures is identification of the invariants. In addition to the `kernel_code_invariants`, we need a number of *auxiliary* invariants about components of the system state.

The primary motivations for exploring model checking as an alternate proof strategy are to increase the automation and eliminate the need for auxiliary invariants in the proof of the partition property. Our approach is to incorporate the information in `kernel_code_invariants` in constructing the abstract system to model check. Taking care to use the same level of abstraction for the kernel code in both the deductive and algorithmic proofs, it turns out that the stack pointer invariant is the only invariant required for reducing the problem to a model checking proof (cf. Section 5).

In the remainder of the paper, we describe verification of the partitioning architecture with respect to the two approaches outlined above. The deductive verification uses rewriting and the PVS decision procedures, and the algorithmic proof uses the PVS model checking capability. As of this reporting, the PVS proof using decision procedures is complete and the PVS model checking proof is nearly complete. This paper is organized as follows. Section 2 outlines the partitioning architecture and introduces the version of the partitioning property that we use. Section 4 describes verification of the property using decision procedures. Section 5 presents the corresponding discussion for the finite state verification, including one additional wrinkle: the model checking proof is not complete until the abstracted system is shown to be a conservative abstraction of the original system. This step is fairly easy in our case because the abstraction is essentially conservative by construction. In both approaches, completing the verification requires proof that the actual kernel code satisfies the `kernel_code_invariants`. The last section discusses these additional proofs.

2 Partitioning Architectures and Their Properties

The purpose of partitioning is fault containment: a failure in one partition must not be allowed to propagate and cause failure in another partition. Although propagation of failures may be due to physical faults in the underlying hardware, this type of propagation is best handled by fault-tolerant mechanisms. The intent of partitioning in our work is to control the effect of faults in the design and implementation of tasks in partitions that share resources. As this focus suggests, the

design choices for partitioning interact with those for providing operating system services. The architecture we study uses a "virtual machine" approach, shown in Figure 1, that relies for enforcement on the kernel and its supporting hardware.

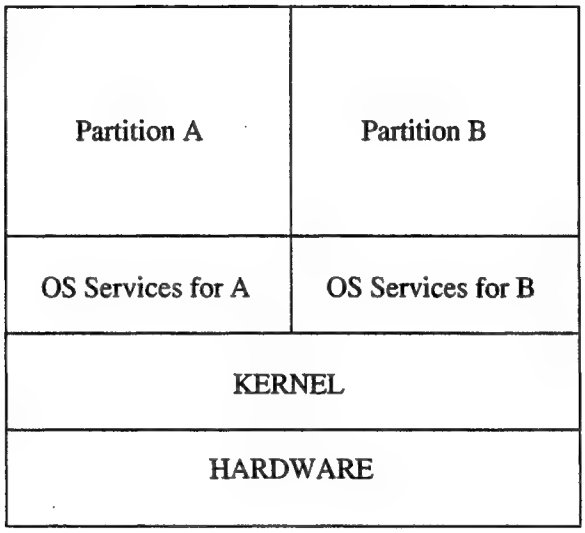


Figure 1: A General Partitioning Architecture

There are two kinds of partitioning: *spatial* and *temporal.* Spatial partitioning is concerned with memory separation, while temporal partitioning is concerned with the independence of timed events in separate tasks. The property we verify involves spatial partitioning. The basic concern of spatial partitioning is the possibility that software in one partition might write into the memory of another. Hardware mediation provided by a partition management unit (PMU) is the usual way to guard against violations of spatial partitioning. The basic idea is that the processor has two modes of operation: *user* and *supervisor.* In user mode, all accesses to memory addresses are either checked or translated using tables maintained in the PMU. A *kernel* or layer of operating system software manages the PMU tables so that the memory locations that can be read and written in each partition are disjoint. The kernel also uses the PMU to protect itself from being modified by software in its client partitions, and manages the user/supervisor mode distinctions of the processor to ensure that the mediation provided by the PMU cannot by bypassed.

A typical kernel scheduling algorithm allows the software in one partition to execute for a while, then passes control to another partition and so on; when one partition is suspended and another started, the kernel saves essential information needed to

resume the suspended software where it left off. The technique described here for partitioning is similar to classical time-sharing where partitions may be suspended at arbitrary points and later resumed.

2.1 The Rockwell Collins Partitioning Architecture

The specific architecture we use is shown in Figure 2. The architecture was designed at Rockwell Collins to investigate design alternatives for implementing safe partitioning mechanisms for AAMP-FV, a member of the AAMP family of processors [10]. Since the microcode of AAMP-FV was itself partially formally verified,

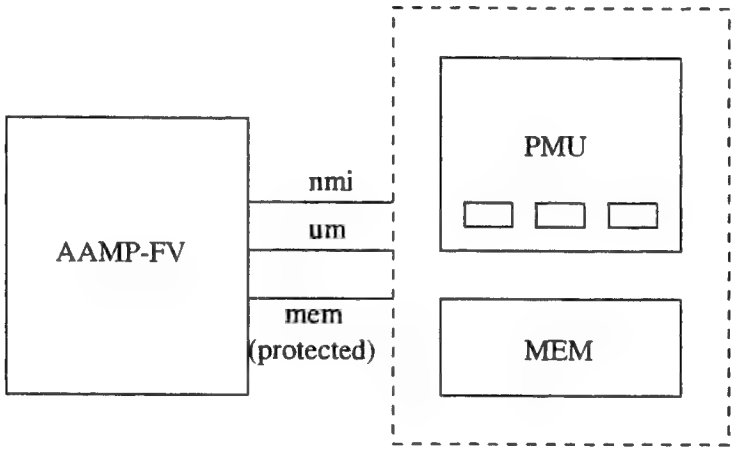


Figure 2: AAMP-FV Partitioning Architecture

this architecture was chosen to provide a nontrivial example of formal verification of a software/hardware system that uses AAMP-FV. The kernel we verify is a piece of AAMP-FV code that manages the PMU tables and the partitioning swaps. The PMU maintains two memory mapped registers for memory partitioning—`base` and `limit`—that define the boundaries of the address space allocated to the current partition. In user mode, the hardware protection in the architecture prohibits writes to addresses outside of this address space.

The state machine shown in Figure 3 is an abstract characterization of the execution behavior of the kernel code. Every transition in the state machine corresponds to execution of an AAMP-FV machine instruction. The sequence of transitions from state `Exec0` to state `User` corresponds to code execution with the processor in executive mode. The sequence of transitions from state `User` to state `Exec0` corresponds to user-task execution. User-to-Exec mode switching occurs by means

of interrupts or trap-like instructions. Exec-to-User mode switching occurs via a special USER instruction. The longest path possible in the kernel code has about 40 instructions.

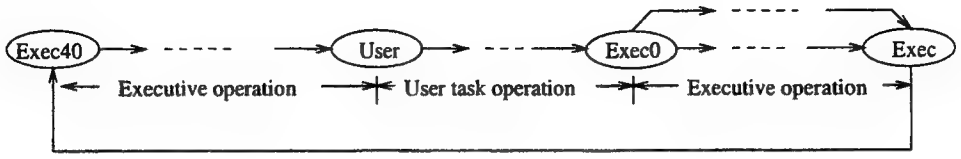


Figure 3: Kernel State Machine

The kernel code also maintains additional *timing* registers in the PMU to implement temporal partitioning mechanisms that ensure that every partition gets its pre-allocated slot of time/service. This ensurance persists even under the uncertainties caused by latency in interrupt handling by the AAMP-FV processor. Since the property we verify concerns memory partitioning, our specification abstracts away most of the details in the kernel code.

2.2 Invariant Performance and Cumulative Invariance

Recently, a number of people have specified partitioning properties of systems formalized on execution traces of the partitioned system [2, 3, 10]. The property partially verified by Rockwell International [10], called *invariant performance* and shown in Figure 4, asserts that "If two kernel-controlled AAMP-FV partition sets have identical initial kernel states and identical states for some partition p, then the execution of the partition sets maintains the equivalence of the states of partition p."

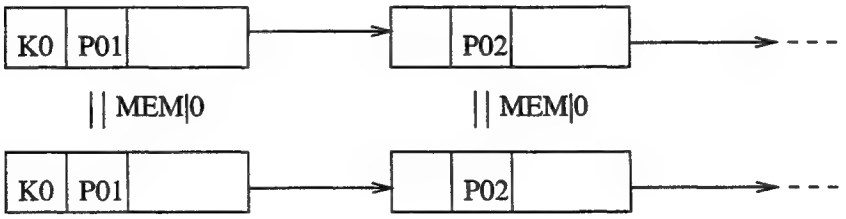


Figure 4: Invariant Performance Property

Note that the invariant performance property, although stated as a condition on memory, ensures certain aspects of temporal partitioning, since it requires confor-

mance after every transition. For example, the scheduling of tasks must be invariant with respect to behavior of software in the partitions. The property we verify, called *cumulative invariance* and shown in Figure 5, is weaker than the Rockwell Collins property. Our property asserts that "If two execution traces starting from the same initial state execute the same sequence of instructions pertaining to a partition p, then the memory projected to the partition p in the two traces after the instruction sequence is complete must be identical." In other words, if the two execution traces are filtered of all transitions except those belonging to `partition p`, the memory projected to partition p must be identical in corresponding states in both traces.

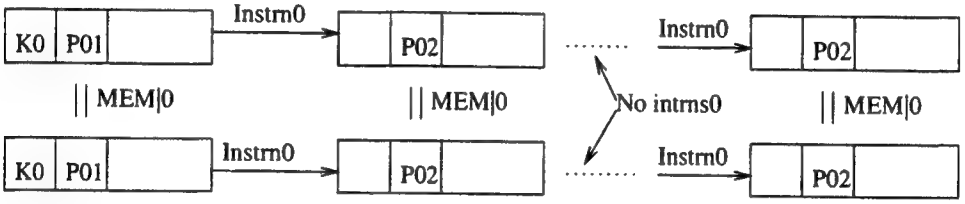


Figure 5: Cumulative Invariance Property

The cumulative invariance property may be proved by induction if the following safety properties are established on the global transition system.

1. Every user-mode-to-user-mode transition must affect only the memory space of the current partition.

2. All other transitions (i.e., all nonuser-mode-to-user-mode transitions) must not affect the user space.

Although we have completed the inductive PVS proof of the cumulative invariance property, we focus in this report on the more instructive proof of the two safety properties.

3 Modeling the System

The system model reflects the standard approach to modeling synchronous systems: a state transition function is defined for every major component of the global system state, in our case, the PMU AAMP-FV, and the connections (CONNS), of which MEMORY is a part. The global transition function `step` is defined as a composition of the component transition functions. Since the system is deterministic, we use a

functional style to define next-state transitions. Excerpts from the definition of the `step` function are shown below.

```
system_state: TYPE = [# processor: macro_state,
pmu : pmu_state,
conns : connections_state #]

step_processor(s,sunprot):macro_state =
 if um(s) AND intreg(s)^6 AND um(s)=um(sunprot) then
   ute_action(sunprot)
 elsif um(s) AND impending_toexec_instruction(s) AND um(s)=um(sunprot) then
   illegal_action(sunprot)
 else next_state(s) ENDIF

step(system): system_state =
  LET conns = conns(system), s = processor(system), p = pmu(system) IN

  % run next instruction with protected memory and PMU-generated NMI
  LET s2 = step_processor(s WITH [(mem):=protectmem(memory(conns),p,conns),
                                  (intreg):=setint6(intreg(s), nmi(conns))],s) IN
  LET mem2 = IF um(s2) THEN
               restorememm(mem(s2),memory(conns),p,conns)
             ELSE mem(s2) ENDIF IN
        .......

   LET p2 = step_pmu(p,memory(conns2),itime) IN
    make_system (s2, p2, conns2)
```

Note, in particular, functions `protect_mem` and `restore_mem`, which model the hardware protection enforced by the PMU on access to memory via the PMU special registers. For example, `protect_mem` provides "read protection" by "zeroing" all read accesses to an address outside the memory partition corresponding to the current execution partition. Similarly, the `restore_mem` function provides "write protection" by restoring writes to the prohibited addresses to their original values. The hardware protection is used only when the system makes a user-mode-to-user-mode transition; in all other cases, protection is provided by kernel code behavior. The structure of the specification is best understood in terms of its functional hierarchy, which is shown in Figure 6.

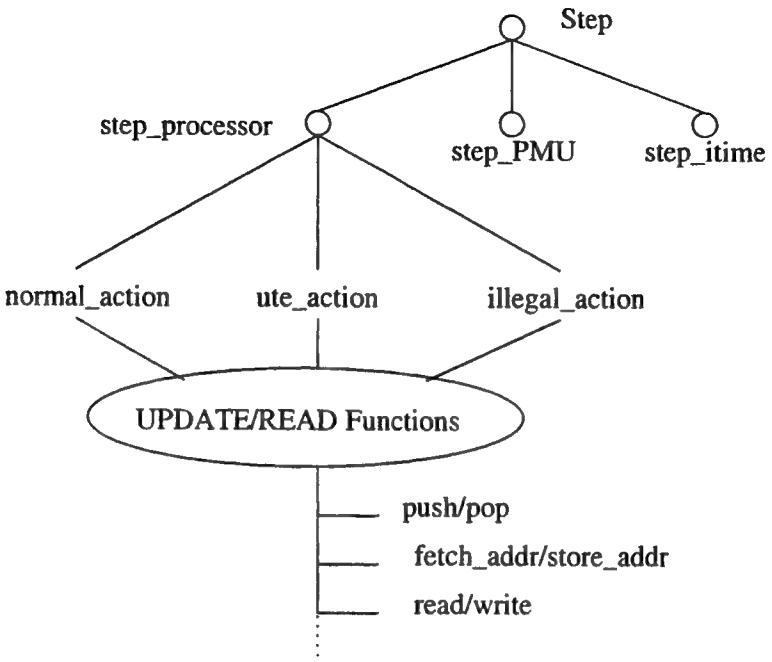


Figure 6: The Specification Hierarchy

4 Verification by Decision Procedures

Correctness Argument

The main idea behind the correctness proof is to use induction to reduce proof of the infinite trace property shown in Figure 5, to a 1-step *noninterference* property whose PVS formalization is shown below. The noninterference property essentially states that every kind of system transition (including executive transitions) that causes a state-change, must modify only the *current* partition.

```
step_property: LEMMA
    FORALL ((system: system_state | invariant(system)), a: address):
      LET conns = conns(system), s = processor(system), p = pmu(system)
      IN in_userspace(a) IMPLIES
         memory(conns(step(system)))(val(a)) = memory(conns)(val(a))
```

We consider four types of transition corresponding to the two possible modes (user and exec) of the current and next states. In user-mode-to-user-mode transitions,

the hardware protection guarantees safe memory partitioning provided that the PMU base and limit registers maintain the correct address boundary information for the current partition. This constraint is characterized as the *PMU invariant.* The correctness argument relies on the preservation of the PMU invariant by all other modes of transitions: in user mode, the base and limit (memory-mapped) registers must correspond to the (constant) base and limit addresses associated with the current user partition. The PVS formalization of the PMU invariant is shown below.

```
pmu_mem_cond(sys: system_state): bool =
  LET mem = memory(conns(sys)) IN
   um(conns(sys)) IMPLIES
      base(curr(mem), mem)  = sys`pmu`base  &
      limit(curr(mem), mem) = sys`pmu`limit
```

To allow the inductive proof to go through, this invariant must be strengthened significantly by additional invariants. For explanatory purposes, we classify the additional invariants into two categories:

1. *core invariants*: invariants that identify a structure in the executive data space that must be preserved by the kernel and task switching transitions
2. *auxiliary invariants*: invariants that state conditions peripheral to the preservation of the PMU invariant.

Core invariants suggest a partition of the memory address space into a finite number of regions and assert that the contents of a set of selected locations used to store pointers must remain within specified address ranges. For example, one core invariant specifies that the executive stack pointer, `exec_tos`, must remain within the `stack_space` region. Auxiliary invariants state assertions about other locations in the executive data space, asserting either that contents of certain variables must remain unchanged or that a specified relationship between pairs of variables must remain true.

Mechanizing the Proof

Our approach to mechanizing the proof of the PMU invariant uses the PVS decision procedures to provide a *symbolic simulation* strategy that consists of the following steps:

1. provide appropriately strong antecedents for the PMU invariant

2. install auto-rewrite rules to rewrite defined functions to primitive PVS data types

3. invoke GRIND, a PVS defined strategy that repeatedly applies rewriting, propositional simplification, and decision procedures

In principle, if a strong-enough reachability invariant is constructed, the memory partitioning property could be proved in PVS using the three steps outlined above. However, this approach would not work for the PMU invariant because the size of the terms created during the proof would be prohibitively large. Accordingly, we adopt a strategy similar to the three-step process described above, except that we restrict the depth to which the defined functions are rewritten. We rewrite all the way down the functional hierarchy shown in Figure 6, until we reach the level of the update/read functions. In other words, we treat the update/read functions as uninterpreted functions, although we do provide certain lemmas about their behavior to be used by GRIND as rewrite rules. These lemmas assert selective behaviors of the update/read functions that affect memory when the addresses are drawn from the indicated address partitions. For example, the typical lemma `mem_store_register_rule` shown below specifies the behavior of a matched pair of update/read functions with respect to addresses drawn from mutually exclusive regions. The main advantage of treating update and read as partially interpreted functions is that it reduces the size of terms and the number of cases considered by GRIND.

```
mem_store_register_rule: LEMMA
    FORALL (r: register, a: address, b: below(exp2(24))):
       mutually_exclusive(address(b), a) IMPLIES
        mem(store_register(a, r, ms))(b) = mem(ms)(b)
```

With sufficient strengthening, the revised proof strategy now successfully proves the partitioning properties, the invariants, and the lemmas on the update/read functions.

5 Verification by Model Checking

As previously noted, the most time-consuming parts of the deductive proof of cumulative invariance were formulation of the strengthening invariants, and formalization of the behavior of the update/read functions in the abstraction used to decompose the proof. The primary motivation for analyzing the cumulative invariance property with the PVS model checker is to reduce this effort and achieve greater automation.

There are two sources of unboundedness in the design model for the partitioned system: the number of user partitions, and the memory data and address space.

Since the behavior of the system is symmetrical with respect to the main invariant, we need only model the current partition and all other partitions. Accordingly, we reduce the model to two symbolic partitions. Using nondeterminism, system transitions that manipulate the partition number may be correspondingly abstracted.

Abstracting the memory to a finite instance is somewhat more challenging. The deductive proof is helpful in this regard; information accumulated in the invariants constructed for the PVS proof suggests that it is sufficient to consider a finite number of address space partitions. With the exception of the *executive stack pointer*, the contents of all locations may be abstracted to one bit of information indicating whether or not the location is modified. It turns out that we also need a second bit of information reflecting which partition (i.e., current, kernel, or other) is relevant for a given piece of data. Although **push** and **pop** operations may increment and decrement the stack pointer in an unbounded fashion, the system may be abstracted to a finite abstraction that preserves the crucial invariant *as long as* the stack pointer remains invariantly inside one of the identified address partitions of the memory abstraction. Given this stack pointer constraint, which is in fact one of the `kernel_code_invariants`, the cumulative invariance property may be model checked using a conservative abstraction of the original system.

We use the specification hierarchy shown in Figure 6 to construct an abstract definition of the transition function. The property to be verified is similarly abstracted. `Push` and `Pop` functions that affect the stack pointer invariant are modeled as nondeterministic functions that may affect any of the address partitions; however we conjoin the stack pointer invariant with the transition function. The abstract model is verified by invoking the PVS model check command.

6 Discussion

The deductive and algorithmic proofs outlined above must be buttressed with the further proof that the actual kernel code satisfies the `kernel_code_invariants`, including core invariants and four additional kernel code assumptions. The invariants concerning the execution of kernel code instructions may be established in one of two ways. The first approach considers the kernel code one instruction at a time, without any abstraction, and shows that if the given invariants hold prior to execution of a kernel instruction, they are preserved after execution of that instruction. An advantage of this method is that it does not involve setting up further abstractions; a major disadvantage is that it may require further strengthening of the invariant, especially if the kernel code contains loops.

A second approach involves the use of abstraction. We construct a finite state transition system that models an abstract interpreter for kernel code execution by abstracting each instruction in the code with respect to the abstract memory described in Section 5. We then use model checking to verify the `kernel_code_invariants` of this model. The advantage of this approach is that it works well even in the presence of loops. As before, the abstraction yields a finite state system only under the assumption of the stack pointer invariant (cf. Section 5). One way to prove the stack invariant is to construct the abstraction using the predicate abstraction method of [1, 6], with the base predicates specified in terms of the stack pointer and its relation to the boundaries of the stack space region in the memory. In future work we plan to explore the utility of the automated predicate abstraction facility in PVS [7] for verifying the `kernel_code_invariants`.

References

[1] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [4], pages 160–171.

[2] Ben L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In Weinstock and Rushby [9], pages 269–286.

[3] Bruno Dutertre and Victoria Stavridou. A model of non-interference for integrating mixed-criticality software components. In Weinstock and Rushby [9], pages 301–316.

[4] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[5] John Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/~rushby/abstracts/partition`, and `http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/NASA-99-cr209347.pdf`; also issued by the FAA.

[6] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[7] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [4], pages 443–454.

[8] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.

[9] Charles B. Weinstock and John Rushby, editors. *Dependable Computing for Critical Applications—7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, San Jose, CA, January 1999. IEEE Computer Society.

[10] Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In Weinstock and Rushby [9], pages 287–300.

In E. A. Emerson and A. P. Sistla, editors, Computer-Aided Verification, CAV '2000, volume 1855 of Lecture Notes in Computer Science, pages 508-520, Chicago, IL, July 2000. ©Springer-Verlag

Verification Diagrams Revisited: Disjunctive Invariants for Easy Verification*

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

*This research was supported by DARPA through USAF Rome Laboratory Contract F30602-96-C-0204 and USAF Electronic Systems Center Contract F19628-96-C-0006, and by the National Science Foundation contract CCR-9509931.

Abstract

I describe a systematic method for deductive verification of safety properties of concurrent programs. The method has much in common with the "verification diagrams" of Manna and Pnueli [17], but derives from different intuitions. It is based on the idea of strengthening a putative safety property into a disjunction of "configurations" that can easily be proved to be inductive. Transitions among the configurations have a natural diagrammatic representation that conveys insight into the operation of the program. The method lends itself to mechanization and is illustrated using a simplified version of an example that had defeated previous attempts at deductive verification.

1 Introduction

In 1997, Shmuel Katz, Patrick Lincoln and I presented an algorithm for Group Membership together with a detailed, but informal proof of its correctness [14]. Shortly thereafter, our colleague Shankar and, independently, Sadie Creese and Bill Roscoe of Oxford University, noted that the algorithm is flawed when the number of nonfaulty processors is three. Model checking a downscaled instance can be effective in finding bugs (that is how Creese and Roscoe found the problem in our algorithm [8]), but true assurance for a potentially infinite-state n-process algorithm such as this seems to require (mechanically checked) deductive methods—either direct proof or justification of an abstraction that can be verified by algorithmic means. Over the next year or so, Katz, Lincoln and I each made several attempts to formalize and mechanically verify a corrected version of the algorithm using the PVS verification system [19]. On each occasion, we were defeated by the number and complexity of the auxiliary invariants needed, and by the "case explosion" that bedevils deductive approaches to formal verification.

Eventually, I stumbled upon the method presented in this paper and completed the verification in April 1999 [23]. This new method made the verification not merely possible, but easy, and it provides a visual representation that conveys considerable insight into the operation of the algorithm. Holger Pfeifer of the University of Ulm was subsequently able to use the method to verify a related but much more complicated group membership algorithm [21] used in the Time Triggered Architecture for critical real-time control [15]

I later discovered that my method has much in common with the "verification diagrams" introduced by Manna and Pnueli [17], and subsequently generalized by Manna and several colleagues [5, 7, 10, 16]. However, the intuition that led to my method is rather different than that for verification diagrams, as is the way I approach its mechanization. I hope that by revisiting these methods from a slightly different perspective, I will help others to see their value and to investigate their application to new problems.

I describe my method in the next section and present an example of its application in the one after that. The final section compares the method with verification diagrams and with other techniques and provides conclusions and suggestions for further work.

2 The Method

Concurrent systems are modeled as nondeterministic automata over possibly infinite sets of states. Given set of states S, initiality predicate I on S, and transition relation T on S, a predicate P on S is *inductive* for $\mathcal{S} = (S, I, T)$ if

$$I(s) \supset P(s)^1 \tag{1}$$

and

$$P(s) \wedge T(s,t) \supset P(t). \tag{2}$$

[1]Formulas are implicitly universally quantified in their free variables; the horseshoe symbol $\supset$ denotes logical implication.

The *reachable states* are those characterized by the smallest (ordered by implication) inductive predicate R on $\mathcal{S}$. A predicate G is an *invariant* or *safety property* if it is larger than R (i.e., includes all reachable states). The focus here is on safety (as opposed to liveness) properties, so we do not need to be concerned with the acceptance criterion on the automaton $\mathcal{S}$.

The deductive method for verifying safety properties attempts to establish that a predicate G is invariant by showing that it is inductive—i.e., we attempt to prove the verification conditions (1) and (2) with G substituted for P. The problem, of course, is that many safety properties are not inductive, and must be strengthened (i.e., replaced by a smaller property) to make them so. Typically, this is done by conjoining additional predicates in an incremental fashion, so that G is replaced by

$$G^i_\wedge = G \wedge G_1 \wedge \cdots \wedge G_i \tag{3}$$

until an inductive $G^m_\wedge$ is found. This process can be made systematic, but is always tedious. In one well-known example, 57 such strengthenings were required to verify a communications protocol [12]; each G_{i+1} was discovered by inspecting a failed proof for inductiveness of $G^i_\wedge$, and the process consumed several weeks.

Some improvements can be made in this process: static analysis [4] and automated calculations of (approximations to) fixpoints of weakest preconditions or strongest postconditions [5] can discover many useful invariants that can be used to seed the process as $G_1, \ldots, G_i$. Nonetheless, the transformation of a desired safety property into a provably inductive invariant remains the most difficult and costly element in deductive verification, and systematic methods are sorely needed.

The method proposed here is based on strengthening a desired safety property with a *disjunction* of additional predicates, rather than the *conjunction* appearing in (3). That is, we construct

$$G^m_\vee = G \wedge (G_1 \vee \cdots \vee G_m)$$

instead of $G^m_\wedge$. Obviously, this can be rewritten as follows

$$G^m_\vee = (G \wedge G_1) \vee \cdots \vee (G \wedge G_m).$$

Rather than form each disjunct as a conjunction $(G \wedge G_i)$, it is generally preferable to use

$$G^m_\vee = G'_1 \vee \cdots \vee G'_m \tag{4}$$

and then prove $G'_i \supset G$ for each G'_i. The subexpressions G'_i are referred to as *configurations*, and the indices i as *configuration indices*.

Observe that in the construction of $G^m_\wedge$, each G_i must be an invariant (the very property we are trying to establish), and that the inadequacy of $G^i_\wedge$ only becomes apparent through failure of the attempted proof of its inductiveness—and proof of the putative inductiveness of $G^{i+1}_\wedge$ must then start over.[2] In contrast, the configurations used in construction of $G^m_\vee$ need not themselves be invariants, and can be discovered

[2]PVS attempts to lessen the amount of rework that must be performed in this situation by allowing conjectures to be modified during the course of a proof; such proofs are marked provisional until a final "clean" verification is completed.

in a rather systematic manner. To see this, first suppose that $G_\vee^m$ is inductive, and consider the proof obligations needed to establish this fact. Instantiating (2) with $G_\vee^m$ of (4) and case-splitting across the configurations, we will need to prove a verification condition of the following form for each configuration index i:

$$G_i'(s) \wedge T(s,t) \supset G_1'(t) \vee \cdots \vee G_m'(t).$$

We can further case-split on the right of the implication by introducing predicates $C_{i,j}(s)$ called *transition conditions* such that, for each configuration index i

$$\forall s \in S : \bigvee_j C_{i,j}(s) \tag{5}$$

(here j ranges over the indices of the transition conditions for configuration G_i') and

$$G_i'(s) \wedge T(s,t) \wedge C_{i,j}(s) \supset G_j'(t) \tag{6}$$

for each transition condition $C_{i,j}$ of each configuration G_i'. Note that some of the $C_{i,j}$ may be identically *false* (so that the proof obligation (6) is vacuously true for this case) and that it is not necessary that the $C_{i,j}$ for different j be disjoint.

This construction can be represented in a diagrammatic form called a *configuration diagram* such as that shown several pages ahead in Figure 1. Here, each vertex represents a configuration and is labeled with the name of the corresponding formula G_i' and each arc represents a non-*false* transition condition and is labeled with a phrase that suggests the corresponding predicate. To verify the diagram, we need to show that the initiality predicate implies some disjunction of configurations

$$I(s) \supset G_1'(s) \vee \cdots \vee G_m'(s) \tag{7}$$

(typically there is just a single *starting configuration*), that each configuration implies the desired safety property

$$G_1'(s) \vee \cdots \vee G_m'(s) \supset G(s), \tag{8}$$

that the disjunction of the transition conditions leaving each configuration is *true* (i.e., (5)), and that the transition relation indeed relates the configurations in the manner shown in the diagram (i.e., the verification conditions (6)). Notice that this is just a new way of organizing a traditional deductive invariance proof (i.e., the proof obligations (5)–(8) imply (1) and (2) with G substituted for P). And although a configuration diagram has some of the character of an abstraction, its verification involves only the original model, and no new verification principles are involved.

The previous discussion assumed we already had a configuration diagram; in practice, the diagram is constructed incrementally in the course of the proof. To construct a configuration diagram, we start by inventing a starting configuration and checking that it is implied by the initiality predicate and implies the safety property (i.e., proof obligations (7) and (8)). Then, by contemplation of the algorithm (the guard predicates and other case-splits in the specification are good guides here), we invent some transition conditions for the starting configuration and check that their disjunction is

true (i.e., proof obligation (5)). For each transition condition, we symbolically simulate a step of the algorithm from the starting configuration, under that condition. The result of symbolic simulation becomes a new configuration (and implicitly discharges proof obligation (6) for that case)—unless we recognize it as a variant of an existing configuration, in which case we must explicitly discharge proof obligation (6) by proving that the result of symbolic simulation implies the existing configuration concerned (sometimes it may be necessary to generalize an existing configuration, in which case we will need to revisit previously-proved proof obligations involving this configuration to ensure that they are preserved by the generalization). We also check that each new or generalized configuration implies the safety property (i.e., proof obligation (8)). This process is repeated for each transition condition and each new configuration until the diagram is closed. The creative steps are the selection of transition conditions, and recognition of new configurations as variants of existing ones. Neither of these is hard, given an informal understanding of the algorithm being verified, and the resulting diagram not only verifies the desired safety property (once all its proof obligations are discharged), but it also serves to explain the operation of the algorithm in a very effective way. Bugs in the algorithm, or unfortunate choices of configurations or of transition conditions, will be manifested as difficulty in closing the diagram (typically, the result of a symbolic simulation step will not imply the expected configuration). As with most deductive methods, it can be tricky to distinguish between these causes of failure.

3 An Example: Group Membership

A simplified version of the group membership algorithm mentioned earlier [14] will serve as an example. There are n processors numbered $0, 1, \ldots, n-1$ connected to a broadcast bus; a distributed clock synchronization algorithm (not discussed here) provides a global clock that ticks off "slots" $0, 1, 2, \ldots$ In slot i it is the turn of processor $i \bmod n$ to broadcast. The broadcast contains a message, not considered here, and the `ack` bit of the broadcasting processor, which is described below. Processors may be faulty or nonfaulty; those that are faulty may be *send-faulty*, *receive-faulty*, or both. A processor that is send-faulty will fail to send its broadcast message in its first slot after it becomes faulty; thereafter it may or may not broadcast in its slots. A processor that is receive-faulty will fail to receive the first broadcast from a nonfaulty processor after it becomes faulty; thereafter it may or may not receive broadcasts. Notice that faults affect only communications: a faulty processor still executes the algorithm correctly; additional elements in the full protocol suite ensure that other kinds of faults are manifested as "fail silence," which appears to the algorithm described here as a combined send- and receive-fault in the processor concerned.

Each processor maintains a *membership set* which contains all and only the processors that it believes to be nonfaulty. Processors broadcast in their slots only if they are in their own membership sets. The goal of the algorithm is to maintain accurate membership sets: all nonfaulty processors should have the same membership sets (this is the *agreement* property) and those membership sets should contain all the nonfaulty processors and at most one faulty one (this is the *validity* property; it is necessary to al-

low one faulty processor in the membership because it takes time to diagnose a fault). These safety properties must be ensured subject to the *fault arrival hypothesis* that faults do not arrive closer than n slots apart. Initially all processors are nonfaulty, their membership sets contain all processors, and their ack bits are *true*.

The algorithm is a synchronous one: in each slot one processor broadcasts and all the other processors expect to receive its message, provided the broadcaster is in their membership sets. Receivers set their ack bits to *true* in each slot iff they receive an expected message. In addition, they remove the broadcaster from their membership sets if they fail to receive an expected message (on the interim assumption that the broadcaster must have been send-faulty). A receiver that subsequently receives a message carrying ack *false* when its own ack is also *false* knows that it made the correct decision in this case (since the current broadcaster also missed the previous expected message), but one that receives ack *true* realizes that it must have been receive-faulty (since the current broadcaster did receive the message) and removes itself from its own membership; a receiver that fails to receive an expected message when its ack bit is *false* also removes itself from its own membership (because it has missed two expected messages in a row, which is consistent with the fault arrival hypothesis only if that processor is itself receive-faulty); a receiver that receives a message with ack *false* when its own ack bit is *true* removes the broadcaster from its membership (since the broadcaster must have been receive-faulty on the previous broadcast). Processors that remove themselves from their own membership remain silent when it is their turn to broadcast—thereby communicating their self-diagnosed receive-faultiness to the other processors.

Formally, we let $\mathtt{mem}(p)$ and $\mathtt{ack}(p)$ denote the membership set and ack bit of processor p. Note that processor p has access to its own mem and ack, and can also read the value of $\mathtt{ack}(b)$, where $b = i \bmod n$ and i is the current slot number, because this is sent in the message broadcast in that slot.

Initiality predicate: $\mathtt{mem}(p) = \{0, 1, \ldots, n-1\}$, $\mathtt{ack}(p) = \mathit{true}$.[3]

The algorithm is specified by two lists of guarded commands: one for the broadcaster and one for the receivers. Primes denote the updated values of the state variables. The current slot is i and the current broadcaster is b, where $b = i \bmod n$.

Broadcaster: Processor b executes the appropriate guarded command from the following list.

(a) $b \in \mathtt{mem}(b)$	$\rightarrow$	$\mathtt{mem}(b)' = \mathtt{mem}(b)$,	$\mathtt{ack}(b)' = \mathit{true}$
otherwise	$\rightarrow$	no change.	

[3] I use the redundant $= \mathit{true}$ because some find that form easier to read.

Receiver: Each processor $p \neq b$ executes the appropriate guarded command from the following list:

The guards (b)–(g) apply when $b \in \mathtt{mem}(p) \wedge p \in \mathtt{mem}(p)$

(b)	$\mathtt{ack}(p) \wedge$ no msg rcvd	$\rightarrow$	$\mathtt{mem}(p)' = \mathtt{mem}(p) - \{b\}$,	$\mathtt{ack}(p)' = \mathit{false}$
(c)	$\mathtt{ack}(p) \wedge \mathtt{ack}(b)$	$\rightarrow$	$\mathtt{mem}(p)' = \mathtt{mem}(p)$,	$\mathtt{ack}(p)' = \mathit{true}$[4]
(d)	$\mathtt{ack}(p) \wedge \neg\mathtt{ack}(b)$	$\rightarrow$	$\mathtt{mem}(p)' = \mathtt{mem}(p) - \{b\}$,	$\mathtt{ack}(p)' = \mathit{true}$
(e)	$\neg\mathtt{ack}(p) \wedge$ no msg rcvd	$\rightarrow$	$\mathtt{mem}(p)' = \mathtt{mem}(p) - \{p\}$	
(f)	$\neg\mathtt{ack}(p) \wedge \neg\mathtt{ack}(b)$	$\rightarrow$	$\mathtt{mem}(p)' = \mathtt{mem}(p)$,	$\mathtt{ack}(p)' = \mathit{true}$
(g)	$\neg\mathtt{ack}(p) \wedge \mathtt{ack}(b)$	$\rightarrow$	$\mathtt{mem}(p)' = \mathtt{mem}(p) - \{p\}$	
otherwise		$\rightarrow$	no change.	

The environment can perform only a single action: it can cause a new fault to arrive—provided no other fault has arrived "recently." Characterization of "recently" is considered below. We let the_mem denote the current set of nonfaulty processors, so that the following specifies arrival of a fault in a previously nonfaulty processor x.

Fault Arrival: $\exists x \in \mathtt{the_mem} : \mathtt{the_mem}' = \mathtt{the_mem} - \{x\}$

The desired safety properties are specified as follows.

Agreement: $p \in \mathtt{the_mem} \wedge q \in \mathtt{the_mem} \supset \mathtt{mem}(p) = \mathtt{mem}(q)$

Validity: $p \in \mathtt{the_mem} \supset \mathtt{mem}(p) = \mathtt{the_mem} \vee \exists x : \mathtt{mem}(p) = \mathtt{the_mem} \cup \{x\}$

The first says that all nonfaulty processors p and q have the same membership sets; the second says that the membership set of a nonfaulty processor p contains all nonfaulty processors, and possibly one faulty one.

The starting configuration is the following: all nonfaulty processors have their ack bits *true* and their membership sets contain just the nonfaulty processors.

Stable: $p \in \mathtt{the_mem} \supset \mathtt{mem}(p) = \mathtt{the_mem} \wedge \mathtt{ack}(p) = \mathit{true}$

It is natural to consider two transition conditions from this configuration: one where a new fault arrives, and one where it does not. In the latter case, the broadcaster will leave its state unchanged (no matter whether its executes command (a) or its "otherwise" case), and the receivers will execute either their command (c) or their "otherwise" case, and leave their states unchanged. The overall effect is to remain in the *stable* configuration. In the case that a new fault arrives, the same transitions as above will be executed but some previously nonfaulty processor x will become faulty, leading to the following configuration.

Latent(x): $x \notin \mathtt{the_mem}$
$\wedge\, p \in \mathtt{the_mem} \cup \{x\} \supset \mathtt{mem}(p) = \mathtt{the_mem} \cup \{x\} \wedge \mathtt{ack}(p) = \mathit{true}$

[4] This case could be absorbed into the "otherwise" clause with no change to the algorithm; however, the structure of the algorithm seems clearer written this way.

There are two transition conditions from *latent*(x): one where x is the broadcaster in the next slot, and one where it is a receiver.

In the former case, x will execute its command (a) while all nonfaulty receivers will note the absence of an expected message and execute their commands (b), leading to the following configuration.

Excluded$_1$(x): $x \notin \texttt{the_mem} \wedge \texttt{mem}(x) = \texttt{the_mem} \cup \{x\} \wedge \texttt{ack}(x) = \mathit{true}$
$\wedge\, p \in \texttt{the_mem} \supset \texttt{mem}(p) = \texttt{the_mem} \wedge \texttt{ack}(p) = \mathit{false}$

In the latter case, a nonfaulty broadcaster will transmit[5] and its message will be received by all nonfaulty receivers, but missed by x, leading to the following configuration.

Missed_rcv(x): $x \notin \texttt{the_mem} \wedge \texttt{mem}(x) = \texttt{the_mem} \cup \{x\} - \{b\} \wedge \texttt{ack}(x) = \mathit{false}$
$\wedge\, p \in \texttt{the_mem} \supset \texttt{mem}(p) = \texttt{the_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

There are four transition conditions from *missed_rcv*(x): one where the next broadcaster is x and it fails to broadcast; one where x does broadcast; one where the next broadcaster is already faulty; and an "otherwise" case. The first of these is similar to the transition from *latent*(x) to *excluded*$_1$(x) and leads to the following configuration.

Excluded$_2$(x): $x \notin \texttt{the_mem} \wedge \texttt{mem}(x) = \texttt{the_mem} \cup \{x\} - \{b\} \wedge \texttt{ack}(x) = \mathit{true}$
$\wedge\, p \in \texttt{the_mem} \supset \texttt{mem}(p) = \texttt{the_mem} \wedge \texttt{ack}(p) = \mathit{false}$

We recognize that *excluded*$_1$(x) and *excluded*$_2$(x) should each be generalized to yield the following common configuration.

Excluded(x): $p \in \texttt{the_mem} \supset \texttt{mem}(p) = \texttt{the_mem} \wedge \texttt{ack}(p) = \mathit{false}$

In the case where x does broadcast, it will do so with `ack` *false*, causing nonfaulty processors to execute their commands (d) and leading directly to the *stable* configuration. The case where the next broadcaster is already faulty causes all nonfaulty processors and processor x to leave their states unchanged (since that broadcaster will not be in their membership sets), thereby producing a loop on *missed_rcv*(x). The remaining case (a broadcast by a nonfaulty processor, executing its command (a)) will cause nonfaulty receivers to execute their commands (c), while x will either miss the broadcast (executing its command (e)), or will discover the *true* `ack` bit on the received message and recognize its previous error (executing its command (g)); in either case, x will exclude itself from its own membership, leading to the following configuration.

Self_diag(x): $x \notin \texttt{the_mem} \wedge x \notin \texttt{mem}(x)$
$\wedge\, p \in \texttt{the_mem} \supset \texttt{mem}(p) = \texttt{the_mem} \cup \{x\} \wedge \texttt{ack}(p) = \mathit{true}$

The transition conditions from this new configuration are those where x is the broadcaster, and those where it is not. In the former case, x will fail to broadcast

[5]Treatment of the case that the next broadcaster is an already-faulty one depends on how fault "arrivals" are axiomatized: in one treatment, a fault is not considered to arrive until it can be manifested (thereby excluding this case); the other treatment will produce a self-loop on *latent*(x) in this case. These details are a standard complication in verification of fault-tolerant algorithms and are not significant here.

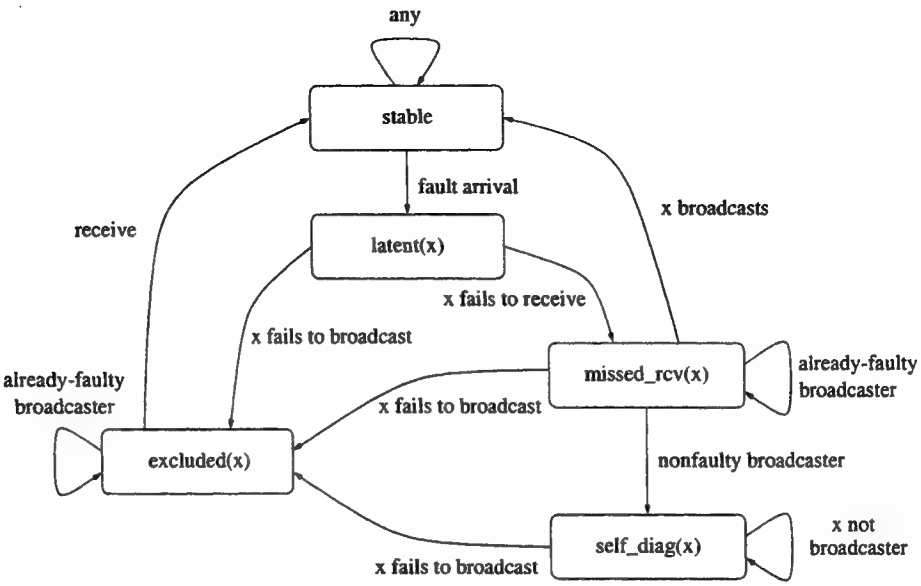


Figure 1: Configuration Diagram for the Group Membership Example

(since it is not in its own membership), causing nonfaulty processors to execute their commands (b) and leading to the configuration *excluded*(x). The other case will cause them to execute their commands (c), or their "otherwise" cases, producing a self-loop on the configuration *self_diag*(x).

The only transitions that remain to be considered are those from configuration *excluded*(x). The transition conditions here are the case where the next broadcaster is already faulty, and that where it is not. The former produces a self-loop on this configuration, while the latter causes all nonfaulty receivers to execute their commands (f) while the broadcaster executes its command (a), leading to a transition to configuration *stable*.

It is easy to see that the initiality predicate implies the *stable* configuration and that all configurations imply the desired safety properties, and so we have now completed construction and verification of the diagram shown in Figure 1. The labels in the vertices of this diagram indicate the corresponding configuration, while the labels on the arcs are intended to suggest the corresponding transition condition. One detail has been glossed over in this construction, however: what about the cases where a new fault arrives while we are still dealing with a previous fault? In fact, this possibility is excluded in the full axiomatization of the fault arrival hypothesis, which states that faults may only arrive when the configuration is *stable* (we then need to discharge trivial proof obligations that all the other configurations are disjoint from this one). We connect this axiomatization of the fault arrival hypothesis with the "real" one that faults must arrive more than n slots apart by proving a bounded liveness property that establishes that the system always returns to a *stable* configuration within n slots of leaving it. This proof requires that configurations are embellished with additional parameters and clauses that

remember the slots on which certain events occurred and count the numbers of self-loop iterations. The details are glossed because they are peripheral to the main concern of this paper; they are present in the mechanized verification of this example using PVS, which is available at `http://www.csl.sri.com/~rushby/cav00.html` and in a paper that describes verification of the full membership protocol [23]. (The full algorithm differs from the simplified version given here in that all faulty processors eventually diagnose their faults and exclude themselves from their own membership; its proof is about four times as long as that presented here).[6]

4 Discussion, Comparison, and Conclusion

The flawed verification of the full membership algorithm in [14] strengthens the desired safety properties, *agreement* and *validity*, with six additional invariants in an attempt to obtain a conjunction that is inductive. Five of these additional invariants are quite complicated, such as the following.

> "If a receive fault occurred to processor p less than n steps ago, then either p is not the broadcaster or $\mathtt{ack}(p)$ is *false* while all nonfaulty q have $\mathtt{ack}(q) = \mathit{true}$, or p is not in its own membership set."

The informal proof of inductiveness of the conjoined invariants is long and arduous, and it must be flawed because the algorithm has a bug in the $n = 3$ case. This proof resisted several determined attempts to correct and formalize it in PVS. In contrast, the approach presented here led to a straightforward mechanized verification of a corrected version of the algorithm.[7] Furthermore, as I hope the example has demonstrated, this approach is naturally incremental, develops understanding of the target algorithm, and yields a diagram that helps convey that understanding to others. In fact, the diagram (or at least its outline) can usually be constructed quite easily using informal reasoning, and then serves as a guide for the mechanized proof.

This approach is strongly related to the verification diagrams and their associated methods introduced by Manna and Pnueli [17]. These were subsequently extended and generalized by Manna with Bjørner, Browne, de Alfaro, Sipma, and Uribe [5, 7, 10, 16]. However, these later methods mostly concern fairness and liveness properties, or extensions for deductive model checking and hybrid systems, and so I prefer to compare my approach with the original verification diagrams. These comprise a set of vertices labeled with formulas and a set of arcs labeled with transitions that correspond to the configurations and transition conditions, respectively, of my method. However, there are small differences between the corresponding notions. First, it appears that verification diagrams have a finite number of vertices, whereas configurations can be finite or infinite in number. The example presented in the previous section

[6] The algorithm presented here is fairly obvious; there is a similarly obvious solution to the full problem (with self-diagnosis) that uses two `ack` bits per message; this clarifies the contribution of [14], which is to achieve full self-diagnosis with only one `ack` bit per message.

[7] The verification was completed on a Toshiba Libretto palmtop computer of decidedly modest performance (75 MHz Pentium with 32 MB of memory).

is a parameterized system with an unbounded parameter n, and most of the configurations are parameterized by an individual x selected from the set $\{0, 1, \ldots, n\}$, yielding an arbitrarily large number of configurations; Skolemization (selection of an arbitrary representative) reduces the number of proof obligations to a finite number. Second, the arcs in verification diagrams are associated with transitions, whereas those in my approach are associated with predicates. It is quite possible that this difference is a natural manifestation of the different examples we have undertaken: those performed with verification diagrams have been asynchronous systems (where each system transition corresponds to a transition by *some* component), whereas I have been concerned with synchronous systems (where each system transition corresponds to simultaneous transitions by *all* components). Thus, in asynchronous systems the transitions suggest a natural analysis by cases, whereas in synchronous systems (especially those, as here, without explicit control) the case analysis must be consciously imposed by selection of suitable transition conditions.

Mechanized support for verification diagrams is provided in STeP [18]: the user proposes a diagram and the system generates the necessary verification conditions. PVS provides no special support for my approach, but its standard mechanisms are adequate because the approach ultimately yields a conventional inductive invariance proof that is checked by PVS in the usual way. As illustrated in the example, the configuration diagram can be constructed incrementally: starting from an existing configuration, the user proposes a transition condition and then symbolically simulates a step of the algorithm (mechanized in PVS by rewriting and simplification); the result either suggests a new configuration or corresponds to (possibly a generalization of) an existing one. Enhancements to PVS that would better support this activity are primarily improvements in symbolic simulation (e.g., faster rewriting and better simplification).

The key to any inductive invariance proof is to find a partitioning of the state space and a way to organize the case analysis so that the overall proof effort is manageable. The method of disjunctive invariants is a systematic way to do this that seems effective for some problem domains. Other recent methods provide comparably systematic constructions for verifications based on simulation arguments: the *aggregation method* of Park and Dill [20] and the *completion functions* of Hosabettu, Gopalakrishnan and Srivas [13] greatly simplify construction of the abstraction functions used in verifying cache protocols and processor pipelines, respectively.

Other methods with some similarity to the approach proposed here are those based on abstractions: typically the idea is to construct an abstraction of the original system that preserves the properties of interest and that has some special form (e.g., finite state) that allows very efficient analysis (e.g., model checking). Methods based on *predicate abstraction* [24] seem very promising [1, 3, 9, 25]. A configuration diagram can be considered an abstraction of the original state machine and it is plausible that it could be generated automatically by predicate abstraction on the predicates that characterize its configurations and transition conditions. However, it is difficult to see how the user could obtain sufficient insight to propose these predicates without constructing most of the configuration diagram beforehand, and it is also questionable whether fully automated theorem proving can construct sufficiently precise abstractions of these fairly difficult examples using current technology.

Such an abstracted system would still have n processes and further reduction would be needed to obtain a finite-state system that could be model checked. Creese and Roscoe [8] do exactly this for the algorithm of [14] using a technique based on a suitable notion of data independence [22]. They use a clever generalization to make the processes of algorithm independent of how they are numbered and are thereby able to establish the abstracted n-process case by an induction whose cases can be discharged by model checking with FDR. This is an attractive approach with much promise, but formal and mechanized justification for the abstraction of the original algorithm still seems quite difficult (Creese and Roscoe provide a rigorous but informal argument).[8]

In summary, the approach presented here is one of a growing number of methods for verifying properties of certain classes of algorithms in a systematic manner. Circumstances in which this approach seems most effective are those where the algorithm concerned naturally progresses through different phases: these give rise to distinct disjuncts G'_i in a disjunctive invariant $G^m_\vee$ but are correspondingly hard to unify within a conjunctive invariant $G^m_\wedge$. Besides those examples already mentioned, the approach has been used successfully by Holger Pfeifer to verify another group membership algorithm [21]: the very tricky and industrially significant algorithm used in the Time Triggered Architecture for safety-critical distributed real-time control [15].

The most immediate targets for further research are empirical and, perhaps, theoretical investigations into the general utility of these approaches. The targets of my approach have all been synchronous group membership algorithms, while the verification diagrams of Manna et al. seem not to have been applied to any hard examples (the verification in STeP of an interesting Leader Election algorithm [6] did not use diagrammatic methods). If practical experience with a variety of different problem types shows the approach to have sufficient utility, then it will be worth investigating provision of direct mechanical support.

Acknowledgments

I am grateful for useful criticisms and suggestions made by the anonymous referees and by my colleagues Jean-Christophe Filliâtre, Patrick Lincoln, Ursula Martin, Holger Pfeifer, N. Shankar, and M. Srivas, and also for feedback received from talks on this material at NASA Langley, SRI, and Stanford.

References

[1] Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [11], pages 146–159.

[8] Verification by abstraction of the communications protocol example mentioned earlier required 45 of the 57 auxiliary invariants used in the direct proof [12].

[2] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996.

[3] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, pages 319–331, Vancouver, Canada, June 1998.

[4] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [2], pages 323–335.

[5] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[6] Nikolaj Bjørner, Uri Lerner, and Zohar Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Second International Conference on Temporal Logic, ICTL'97*, Manchester, England, July 1997.

[7] I. Anca Browne, Zohar Manna, and Henny Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Volume 1026 of Springer-Verlag *Lecture Notes in Computer Science*, pages 484–498, Bangalore, India, December 1995.

[8] S. J. Creese and A. W. Roscoe. TTP: A case study in combining induction and data independence. Technical Report PRG-TR-1-99, Oxford University Computing Laboratory, Oxford, England, 1999.

[9] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [11], pages 160–171.

[10] Luca de Alfaro, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Visual verification of reactive systems. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Volume 1217 of Springer-Verlag *Lecture Notes in Computer Science*, pages 334–350, Enschede, The Netherlands, April 1997.

[11] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[12] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, Volume 1051 of Springer-Verlag *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996.

[13] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Halbwachs and Peled [11], pages 47–59.

[14] Shmuel Katz, Pat Lincoln, and John Rushby. Low-overhead time-triggered group membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, Volume 1320 of Springer-Verlag *Lecture Notes in Computer Science*, pages 155–169, Saarbrücken Germany, September 1997.

[15] Hermann Kopetz and Günter Grünsteidl. TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[16] Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In Armando M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, Volume 1548 of Springer-Verlag *Lecture Notes in Computer Science*, pages 28–41, Amazonia, Brazil, January 1999.

[17] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software: TACS'94*, Volume 789 of Springer-Verlag *Lecture Notes in Computer Science*, pages 726–765, Sendai, Japan, April 1994.

[18] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [2], pages 415–418.

[19] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[20] Seungjoon Park and David L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[21] Holger Pfeifer. Formal verification of the TTA group membership algorithm. In Tommaso Bolognesi and Diego Latella, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification FORTE XIII/PSTV XX 2000*, pages 3–18, Pisa, Italy, October 2000.

[22] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1998.

[23] John Rushby. Formal verification of a low-overhead group membership algorithm, 2000. In preparation.

[24] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[25] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [11], pages 443–454.

Static Analysis for Safe Destructive Updates*

Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

*Sam Owre, John Rushby, and Dave Stringer-Calvert provided valuable input for the design of the evaluation capabilities in PVS, and Sam Owre, Harald Rueß, and Jean-Christophe Filliâtre helped refine the presentation of the ideas.

Abstract

Pure functional programs are more amenable to rigorous mathematical analysis than imperative programs, but are not comparably efficient in terms of space or time. The updates of aggregate data structures, such as arrays, are an important source of space/time inefficiencies in functional programming. Imperative programs can execute such updates in place, whereas the semantics of functional programs require such data structures to be copied. In many functional programs, the execution of updates by copying is redundant and can be carried out destructively. We describe a method for analyzing higher-order, eager functional programs for safe destructive updates. This method has been implemented for the PVS specification language for efficiently animating specifications. Both the update analysis and the proof of correctness are straightforward and can be easily applied to other functional languages.

1 Background

Functional programs, unlike imperative programs, do not perform in-place modifications of aggregate data structures such as arrays. The aggregate update problem for functional programs is that of statically identifying the array updates in a program that can be executed destructively while preserving the semantics of the program. This problem has been widely studied but none of these techniques appears to have actually been implemented in any widely used functional language. We present a simple, efficient, and effective method for the static detection of safe destructive updates in a functional language. The method has been implemented for the functional fragment of the specification language PVS.[1] This fragment is essentially a strongly typed, higher-order language with an eager order of evaluation. The method can be easily adapted to other functional languages. The analysis method is interprocedural. We also outline a proof of the correctness for the introduction of destructive updates.

PVS is a widely used framework for specification and verification. By optimizing functions written in the PVS specification language with safe destructive updates, specifications can be executed for the purposes of animation, validation, code generation, and fast simplification. The technique is presented for a small functional language fragment of PVS, but applies to other functional languages as well.

The concepts are informally introduced using a functional language with booleans, natural numbers, subranges, flat (unnested) arrays over subranges, lambda-abstraction, application, conditionals, and array updates. The technique can be extended to richer languages. A function is defined as $f(x_1, \ldots, x_n) = e$. A few simple examples serve to motivate the ideas. Let `Arr` be an array from the subrange `[0..9]` to the integers. Let A and B be variables of type `Arr`. An array lookup is written as $A(i)$ and array update has the form $A[(c) := d]$. Pointwise addition on arrays $A + B$ is defined to return (a reference to) an array C such that $C(i) = A(i) + B(i)$ for $0 \leq i < 10$. Now consider the function definition

$$f_1(A) = A + A[(3) := 4].$$

When executing $f_1(A)$, the update to A cannot be carried out destructively since the original array is an argument to the + operation. The evaluation of $A[(3) := 4]$ must return a reference to a new array that is a suitably modified copy of the array A.

The implementation of array updates by copying can be expensive in both space and time. It can also be wasteful. Consider the definition

$$f_2(A, i) = A(i) + A[(3) := 4](i).$$

Given an eager, left-to-right evaluation order, the expression $A(i)$ will be evaluated prior to the update $A[(3) := 4]$. Since the original value of A is no longer used in the computation, the array can be updated destructively. Note that if a lazy order of evaluation was being employed, this optimization would depend on the order in which the arguments of + were evaluated. Also, the optimization assumes that array A is not referenced in the context where $f_2(A, i)$ is evaluated. For example, in the definition

$$f_3(A) = A[(3) := f_2(A, 3)],$$

it would be incorrect to execute f_2 so that A is updated destructively since there is a reference to the original A in the context when $f_2(A, 3)$ is evaluated.

[1] The PVS system and related documentation can be obtained from the URL `pvs.csl.sri.com`. The presentation in this paper is for a generic functional language and requires no prior knowledge of PVS.

Next, consider the function definition

$$f_4(A, B) = A + B[(3) := 4].$$

Here, the update to array B can be executed destructively provided A and B are not aliased to the same array reference. This happens, for instance, in the definition

$$f_5(C) = f_4(C, C).$$

In such a situation, it is not safe to destructively update C in the definition of f_4 since the reference to C is active in the definition of f_4 when the update occurs.

The task is that of statically analyzing the definitions of programs involving function definitions such as those of f_1, f_2, f_3, f_4, and f_5, in order to identify those updates that can be executed destructively. Our analysis processes each definition of a function f, and generates a (possibly) destructive analogue f^D of f that contains destructive updates along with the conditions $Lv(f)$ under which it is safe to use f^D instead of f. The analysis when applied to a definition $f(x_1, \ldots, x_n) = e$ produces a definition of the form $f^D(x_1, \ldots, x_n) = e^D$, where some occurrences of updates of the form $e_1[(e_2) := e_3]$ in e have been replaced by destructive updates of the form $e_1[(e_2) \leftarrow e_3]$. The analysis of the examples above should therefore yield

$$\begin{array}{rclcrcl}
f_1^D(A) & = & A + A[(3) := 4] & & Lv(f_1) & = & \emptyset \\
f_2^D(A, i) & = & A(i) + A[(3) \leftarrow 4](i) & & Lv(f_2) & = & \{A \mapsto \emptyset\} \\
f_3^D(A) & = & A[(3) := f_2(A, 3)] & & Lv(f_3) & = & \emptyset \\
f_4^D(A, B) & = & A + B[(3) \leftarrow 4] & & Lv(f_4) & = & \{B \mapsto A\} \\
f_5^D(C) & = & f_4(C, C) & & Lv(f_5) & = & \emptyset
\end{array}$$

We now informally describe the construction of the definition $f^D(x_1, \ldots, x_n) = e^D$ from the definition $f(x_1, \ldots, x_n) = e$. The analysis also generates the table $Lv(f)$ as a partial map from the set of variables $\{x_1, \ldots, x_n\}$ to its powerset such that $x_j \in Lv(f)(x_i)$ if x_j is live in a context (as defined below) within which x_i might be destructively updated. The table $Lv(f)$ can be used to determine whether it is safe to replace $f(a_1, \ldots, a_n)$ by $f^D(a_1, \ldots, a_n)$ in another function definition.

A specific occurrence of an update u of the form $e_1[(e_2) := e_3]$ in e can be identified by decomposing e as $U\{u\}$, where U is an *update context* containing an occurrence of the *hole* $\{\}$, and $U\{u\}$ is the result of filling the hole with the update expression u. In order to determine if the update is safe, we compute

1. The set L of *live* array variables in the update context U. The set L is a conservative estimate of the variables in U that point to active array references in the partially evaluated form of U. These references might be shared with u when the update u in $U\{u\}$ is executed. Let U' represent the partially evaluated form of U at the point where the update u in $U\{u\}$ is to be reduced. All the array references in U' that could possibly clash with those in u must be among those bound to the variables in L. We write the update context as U^L to indicate that the set of array variables L from U are the live variables in the context U' of the evaluation of the update expression u.

2. The set $Ov(e)$ of the array variables in e so that the output array reference of e must be among those bound to the variables in $Ov(e)$.

An update expression $e_1[(e_2) := e_3]$ occurring in an update context U^L for the expression e in a definition $f(x_1, \ldots, x_n) = e$, is *safe* if $L \cap Ov(e_1) = \emptyset$. When this condition holds, it is safe to replace the nondestructive update $e_1[(e_2) := e_3]$ by its destructive counterpart $e_1[(e_2) \leftarrow e_3]$. Informally, this is because the array references that are possible values of e_1 are either those bound to the variables in $Ov(e_1)$ or are freshly created in e_1. The freshly created references in e_1 cannot overlap with those in the context. The above condition ensures that none of the updateable references in e_1 are accessible from the partially evaluated context U' since the references in U' must be among the array references bound to the variables in L.

However, it is possible that one of the references in $Ov(e_1)$ is shared with those in L even when $L \cap Ov(e_1) = \emptyset$ since a single reference might be shared by two distinct variables through *aliasing*. For this purpose, $Lv(f)(x)$ must be constructed so that $L \subseteq Lv(f)(x)$ for each x in $Ov(e_1)$.

An application occurrence $g(a_1, \ldots, a_n)$ is safe in an update context U^L if $Ov(a_i) \cap (L \cup Ov(a_j)) = \emptyset$ for each x_i in the domain of $Lv(g)$ and x_j in $Lv(g)(x_i)$. If this condition holds, then $g(a_1, \ldots, a_n)$ can be safely replaced by $g^D(a_1, \ldots, a_n)$ in the update context U^L since none of the references $Ov(a_i)$ possibly updated in the definition of g^D for $g^D(a_1, \ldots, a_n)$ are live in the context in which they are updated in the definition e^D. The mapping $Lv(f)$ must be defined to satisfy the constraint $L \cup Av(a_j) \subseteq Lv(f)(x)$ for each x in $Ov(a_i)$ such that x_i is in the domain of $Lv(g)$ and x_j in $Lv(g)(x_i)$.

Thus in the examples f_1 to f_5, we have

1. f_1: e is $A + A[(3) := 4]$, $Ov(A)$ is $\{A\}$ and the update context is $(A + \{\})^{\{A\}}$. Since $Ov(A)$ overlaps with the live variables $\{A\}$, the update is not safe.

2. f_2: e is $A(i) + A[(3) := 4](i)$, $Ov(A)$ is $\{A\}$ and the update context is $(A(i) + \{\}(i))^{\emptyset}$. Since the updated variable A is not live in the update context, the update is safe. Note that $Lv(f_2)(A) = \emptyset$.

3. f_3: e is $A[(3) := f_2(A, 3)]$. Here, the update context is $(A[(3) := \{\}])^{\{A\}}$ and A is in the domain of $Lv(f_2)$. Since there is an overlap between $Ov(A)$ and the live variable set $\{A\}$, the occurrence of $f_2(A)$ is unsafe. The update occurring in f_3 can be executed destructively, since there are no live references to A in the update context $(\{\}[(3) := f_2(A, i)])^{\emptyset}$. We then have $Lv(f_3)(A) = \emptyset$.

4. f_4: e is $A + B[(3) := 4]$. Here, $Ov(B)$ is $\{B\}$ and the update context is $(A + \{\})^{\{A\}}$. Since $\{B\} \bigcap \{A\} = \emptyset$, e^D can be written as $A + B[(3) \leftarrow 4]$ in the definition of f_4^D, but note that $Lv(f_4)(B) = \{A\}$.

5. f_5: e is $f_4(C, C)$. The update context here is $\{\}^{\emptyset}$. Here $Lv(f_4)$ maps B to $\{A\}$. Since $Ov(C) = \{C\}$, the analysis detects the aliasing between the binding C for A and C for B. The occurrence of $f_4(C, C)$ is therefore unsafe and cannot be replaced by $f_4^D(C, C)$.

The formal explanation for the destructive update analysis and optimization is the topic of the remainder of the paper. A similar analysis and transformation for safe destructive updates was given independently and earlier by Wand and Clinger [13] for a first-order, eager functional language with flat arrays. In this paper, we go beyond the treatment of Wand and Clinger by

1. Simplifying the presentation of the analysis.

2. Simplifying the proof of correctness through the use of evaluation contexts.

3. Applying the optimization to a richer language with higher-order operations and nested aggregate structures.

4. Carrying out a complexity analysis of the static analysis procedure.

These extensions are the novel contributions of the paper. We have also implemented our method for an executable functional fragment of the widely used specification language PVS. This implementation was carried out in the first part of 1999 and released with PVS2.3 in the Fall of 1999. Functional programs, such as sorting routines, written in this fragment of PVS execute at speeds that are competitive with imperative languages like C, and with comparable space usage.

There is an extremely large body of work on the aggregate update problem. We know of only one language, Clean [10], where such an optimization has actually been implemented. The Clean scheme requires complicated programmer annotations and is actually quite messy. We do not carry out a comparison with languages that are explicitly annotated for the purpose of highlighting updateable data structures. These include the use of state monads [12] and various linear type systems [11]. The method presented here does not rely on any programmer annotations.

There is a large body of work on static analysis applied to the aggregate update problem including that of Hudak [7], Bloss and Hudak [2], Bloss [3], Gopinath and Hennessy [6], and Odersky [8], and Draghicescu and Purushothaman [4]. Most of these analyses apply to lazy functional languages. Laziness complicates the analysis since there is no fixed order of evaluation on the terms as is the case with eager evaluation. Even so, all these analysis methods are complicated, inefficient (exponential), and mostly non-interprocedural. Many of the analyses calculate more information (e.g., reference counts, reading/writing) than is needed for update optimization. The work of Draghicescu and Purushothaman [4] deserves special mention because it contains the key insight that in a language with flat arrays, the sharing of references between a term and its context can occur only through shared free variables.

In summary, in comparison to previous approaches to update analyses, the method given here is simple, efficient, interprocedural, and has been implemented for an expressive functional language. The implementation is competitive with efficient imperative languages. The proof of correctness also matches the simplicity of the analysis.

2 Update Analysis

We describe a small functional language and an update analysis procedure for this language that generates a destructive counterpart to each function definition. The language is strongly typed. Types are exploited in the analysis, but the principles apply to untyped languages as well.

The base types consist of `bool`, `integer`, and *index types* of the form $[0 < \mathit{numeral}]$, where *numeral* is a numeral.[2] The only type constructor is that for function types which are constructed as $[T_1, \ldots T_n \rightarrow T]$ for types $T, T_1, \ldots, T_n$. The language admits subtyping so that $[0 < i]$ is a subtype of $[0 < i']$ when $i \leq i'$, and these are both subtypes of the type `integer`. A function type $[S_1, \ldots, S_n \rightarrow S]$ is a subtype of $[T_1, \ldots, T_n \rightarrow T]$ iff $S_i \equiv T_i$ for $0 < i \leq n$ and S is a subtype of T.

[2]For ease of presentation, we blur the distinction between numbers and numerals.

We do not explain more about the type system and the typechecking of expressions. Readers are referred to the formal semantics of PVS [9] for more details. An array type is a function type of the form $[[0 < i] \rightarrow W]$ for some numeral i and base type W, so that we are, for the present, dealing only with flat arrays.[3]

The metavariable conventions are that W ranges over base types, S and T range over types, x, y, z range over variables, p ranges over primitive function symbols, f and g range over defined function symbols, a, b, c, d, e range over expressions, L, M, N range over sets of array variables.

The expression forms in the language are

1. Constants: Numerals and the boolean constants `TRUE` and `FALSE`.
2. Variables: x
3. Primitive operations p (assumed to be nondestructive) and defined operations f.
4. Abstraction: $(\lambda(x_1 : T_1, \ldots, x_n : T_n) : e)$, is of type $[T_1, \ldots, T_n \rightarrow T]$, where e is an expression of type T given that each x_i is of type T_i for $0 < i \leq n$. We often omit the types $T_1, \ldots, T_n$ for brevity.
5. Application: $e(e_1, \ldots, e_n)$ is of type T where e is an expression of type $[T_1, \ldots, T_n \rightarrow T]$ and each e_i is of type T_i.
6. Conditional: `IF` e_1 `THEN` e_2 `ELSE` e_3 is of type T, where e_1 is an expression of type `bool`, and e_2, and e_3 are expressions of type T.
7. Update: $e_1[(e_2) := e_3]$, where e_1 is of array type $[[0 < i] \rightarrow W]$, e_2 is an expression of type $[0 < i]$, and e_3 is an expression of type W. The destructive calculus contains a destructive update expression $e_1[(e_2) \leftarrow e_3]$.

A program is given by a sequence of function definitions where each function definition has the form $f(x_1 : T_1, \ldots, x_n : T_n) : T = e$. The body e of the definition of f cannot contain any functions other than f, the primitive operations, and the previously defined functions in the sequence. The body e cannot contain any free variables other than those in $\{x_1, \ldots, x_n\}$.

A type is *mutable* if it is an array type or is a function type whose range type is mutable. A variable is mutable if its type is mutable. $Mv(a)$ is the set of all mutable free variables of a, $Ov(a)$ is the set of output variables in a, and $Av(a)$ is the set of active variables in the output of a. These will be defined more precisely below.

An *update context* U^L is an expression U containing a single occurrence of a hole $\{\}$ where L is the set of free variables from U that are *live*, i.e., point to references in the context when the expression filling the hole is evaluated. An update context U^L has one of the forms

1. $\{\}^{\emptyset}$.
2. $V^M(e_1, \ldots, e_n)$, if $L = M \cup Mv(e_1) \cup \ldots \cup Mv(e_n)$.

[3]The language used here is similar to that employed by Wand and Clinger [13] but with the important inclusion of higher-order operations and lambda-abstraction. We allow arrays to be built by lambda-abstraction, whereas Wand and Clinger use a `NEW` operation for constructing arrays.

3. $e(e_1, \ldots, e_{j-1}, V^M, e_{j+1}, \ldots, e_n)$, if $L = M \cup Av(e) \cup Av(e_1) \cup \ldots \cup Av(e_{j-1}) \cup Mv(e_{j+1}) \cup \ldots \cup Mv(e_n)$. Note that with respect to the hole, $Av(a)$ is used for "already evaluated" expressions a, whereas $Mv(b)$ is used for "unevaluated" expressions b.

4. IF V^M THEN e_2 ELSE e_3, if $L = M \cup Mv(e_2) \cup Mv(e_3)$.

5. IF e_1 THEN V^L ELSE e_3.

6. IF e_1 THEN e_2 ELSE V^L.

7. $V^L[(e_2) := e_3]$ (and $V^L[(e_2) \leftarrow e_3]$ in the destructive fragment).

8. $e_1[(V^M := e_3]$ (and $e_1[(V^M) \leftarrow e_3]$), if $L = M \cup Mv(e_1) \cup Mv(e_2)$.

9. $e_1[(e_2) := V^M]$ (and $e_1[(e_2) \leftarrow V^M]$), if $L = M \cup Mv(e_1)$.

Let $\gamma(e)$ represent the result of repeatedly replacing destructive updates $e_1[e_2 \leftarrow e_3]$ in e by corresponding nondestructive updates $\gamma(e_1)[\gamma(e_2) := \gamma(e_3)]$, and destructive applications $g^D(a_1, \ldots, a_n)$ by $g(\gamma(a_1), \ldots, \gamma(a_2))$. To obtain the destructive definition $f^D(x_1, \ldots, x_n) = e^D$ and the liveness table $Lv(f)$, from the definition $f(x_1, \ldots, x_n) = e$, we construct e^D so that $\gamma(e^D) \equiv e$ and e^D is safe. An expression e^D is *safe* if

1. Every occurrence of $e_1[(e_2) \leftarrow e_3]$ in e^D within an update context U^L (i.e., $e \equiv U^L\{e_1[(e_2) := e_3]\}$) satisfies $Ov(e_1) \cap L = \emptyset$ and $L \subseteq Lv(f)(x)$ for each variable x in $Ov(e_1)$.

2. Every occurrence of a nondestructive function application $g(a_1, \ldots, a_n)$ in e within an update context U^L (i.e., $e \equiv U^L\{g(a_1, \ldots, a_n)\}$) satisfies $Ov(a_i) \cap (L \cup Av(a_j)) = \emptyset$ for each x_i in the domain of $Lv(g)$ and $y_i \in Lv(g)(x_i)$. Furthermore, $L \cup Av(a_j) \subseteq Lv(f)(x)$ for each variable x in $Ov(a_i)$ for x_i in the domain of $Lv(g)$ and $x_j \in Lv(g)(x_i)$.

Given the destructive body e^D for f^D, the liveness table $Lv(f)$ can be computed to satisfy the constraints given by e^D. Note that f could be recursively defined and the operation g in the definition of safety could be a recursive occurrence of f, i.e., the function being defined. The liveness table $Lv(f)$ can be constructed to satisfy the constraints either by solving the set constraints [1] or directly by a fixed point iteration where the above constraints are used to compute $Lv(f)$ cumulatively by starting with $Lv^0(f)(x_i) = \perp$ for $0 < i \leq n$ until $Lv^{k+1}(f) = Lv^k(f)$. Termination is guaranteed because the set $Lv(f)(x_i)$ for $0 < i \leq n$ can have at most $n - 1$ elements from $\{x_1, \ldots, x_n\}$. The definition e^D that is computed for f^D is the one that is carried out with $Lv(f) = Lv^k(f)$.[4] We are glossing over one important point. The set of variables $Ov(f^D(a_1, \ldots, a_n))$ is needed in the computation of $Lv(f)$ and has to be computed iteratively along with $Lv(f)$.

The definitions of the operations Av, Mv, and Ov are given below. $Mv(a)$ is just the set of mutable free variables of a. The set of output variables of an expression a of array type is computed by $Ov(a)$. If a is evaluated to a' in an environment σ that maps variables to values, then the references in $\sigma(Ov(a))$ must include all the references in a' that are accessible from σ. Thus $Ov(a)$ could be conservatively calculated as the set of all mutable variables in a, i.e., $Mv(a)$, but the analysis below is more precise. The auxiliary function $Ovr(a)$ computes a lambda-abstracted set of variables $(\lambda(x_1, \ldots, x_n) : S)$ for a defined function or a lambda-abstraction in the function position of an

[4]We are not dealing with mutually recursive functions here, but the analysis can be extended to include such definitions.

application. This yields a more precise estimate of the set of output variables. For example, if $X + Y$ is defined as $(\lambda(x : [0 < i]) : X(x) + Y(x))$, then $Ovr(X + Y) = (\lambda(x, y) : \emptyset)(X, Y) = \emptyset$.

$$
\begin{aligned}
Ovr(a) &= \emptyset, \text{ if } a \text{ is not of mutable type} \\
Ovr(x) &= \{x\}, \text{ if } x \text{ is of mutable type} \\
Ovr(f) &= Ovr((\lambda(x_1, \ldots, x_n) : e)), \text{ where } e \text{ is the body of } f \\
Ovr(a(a_1, \ldots, a_n)) &= Ovr(a)(Ov(a_1), \ldots, Ov(a_n)) \\
Ovr((\lambda(x_1, \ldots, x_n) : e)) &= (\lambda(x_1, \ldots, x_n) : Ov(e)) \\
Ovr(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3) &= Ov(e_2) \cup Ov(e_3) \\
Ovr(a_1[(a_2) := a_3]) &= \emptyset \\
Ovr(a_1[(a_2) \leftarrow a_3]) &= Ov(a_1) \\
Ov(a) &= S - \{x_1, \ldots, x_n\}, \text{ if } Ovr(a) = (\lambda(x_1, \ldots, x_n) : S) \\
Ov(a) &= Ovr(a), \text{ otherwise} \\
(\lambda(x_1, \ldots, x_n) : S)(S_1, \ldots, S_n) &= (S - \{x_1, \ldots, x_n\}) \cup \bigcup \{S_i | x_i \in S\} \\
S(S_1, \ldots, S_n) &= S \cup S_1 \cup \ldots \cup S_n
\end{aligned}
$$

The set $Av(a)$ of variables returns the active variables and is used to keep track of the variables that point to active references in already-evaluated expressions. The set $Av(a)$ includes $Ov(a)$ but also contains mutable variables that occur in closures.

$$
\begin{aligned}
Av(a) &= \emptyset, \text{ if } a \text{ is not of function or mutable type} \\
Av(x) &= \{x\}, \text{ if } x \text{ is of function or mutable type} \\
Av(f) &= \emptyset \\
Av(a(a_1, \ldots, a_n) &= Ovr(a)(Av(a_1), \ldots, Av(a_n)) \\
Av((\lambda(x_1, \ldots, x_n) : e)) &= Mv((\lambda(x_1, \ldots, x_n) : e)) \\
Av(\texttt{IF } e_1 \texttt{ THEN } e_2 \texttt{ ELSE } e_3) &= Av(e_2) \cup Av(e_3) \\
Av(a_1[(a_2) := a_3]) &= \emptyset \\
Av(a_1[(a_2) \leftarrow a_3]) &= Av(a_1)
\end{aligned}
$$

3 Operational Semantics

We present operational semantics for the languages with and without destructive updates. We then exhibit a bisimulation between evaluation steps in the two semantics. The concepts used in defining the operational semantics are quite standard, but we give the details for the language used here.

The expression domain is first expanded to include

1. Explicit arrays: $\#(e_0, \ldots, e_{n-1})$ is an expression representing an n-element array.
2. References: $ref(i)$ represents a reference to reference number i in the store. Stores appear in the operational semantics.

A *value* is either a boolean constant, an integer numeral, a closed lambda abstraction $(\lambda x_1, \ldots, x_n : e)$ or a reference $ref(i)$. The metavariable v ranges over values.

An *evaluation context* [5] E is an expression with an occurrence of a hole [] and is of one of the forms

1. [].
2. $E'(e_1, \ldots, e_n)$.
3. $v(v_1, \ldots, v_{j-1}, E', e_{j+1}, \ldots, e_n)$.
4. IF E' THEN e_2 ELSE e_3.
5. IF TRUE THEN E' ELSE e_3.
6. IF FALSE THEN e_2 ELSE E'.
7. $e_1[(E') := e_3]$.
8. $e_1[(v_2) := E']$.
9. $E'[(v_2) := v_3]$.

A *redex* is an expression of one of the following forms

1. $p(v_1, \ldots, v_n)$.
2. $f(v_1, \ldots, v_n)$.
3. $(\lambda(x : [0 < n]) : e)$.
4. $(\lambda(x_1, \ldots, x_n) : e)(v_1, \ldots, v_n)$.
5. $\#(v_0, \ldots, v_{n-1})$.
6. IF TRUE THEN e_1 ELSE e_2.
7. IF FALSE THEN e_1 ELSE e_2.
8. $ref(i)[(v_2) := v_3]$.

A *store* is a mapping from a reference number to an array value. A store s can be seen as a list of array values $[s[0], s[1], \ldots,]$ so that $s[i]$ returns the $(i+1)$th element of the list.

A *reduction* transforms a pair consisting of a redex and a store. The reductions corresponding to the redexes above are

1. $\langle p(v_1, \ldots, v_n), s\rangle \rightarrow \langle v, s\rangle$, if the primitive operation p when applied to arguments $v_1, \ldots, v_n$ yields value v.
2. $\langle f(v_1, \ldots, v_n), s\rangle \rightarrow \langle [v_1/x_1, \ldots, v_n/x_n](e), s\rangle$, if f is defined by $f(x_1, \ldots, x_n) = e$.
3. $\langle (\lambda(x : [0 < n]) : e), s\rangle \rightarrow \langle \#(e_0, \ldots, e_{n-1}), s\rangle$, where $e_i \equiv (\lambda(x : [0 < n]) : e)(i)$, for $0 \leq i < n$.

4. $\langle(\lambda(x_1 : T_1, \ldots, x_n : T_n) : e)(v_1, \ldots, v_n), s\rangle \rightarrow \langle[v_1/x_1, \ldots, v_n/x_n](e), s\rangle$.

5. $\langle\#(v_0, \ldots, v_{n-1}), s\rangle \rightarrow \langle ref(m), s'\rangle$, where $s \equiv [s[0], \ldots, s[m-1]]$ and $s' \equiv s \circ [\#(v_0, \ldots, v_{n-1})]$.

6. $\langle$IF TRUE THEN e_1 ELSE $e_2, s\rangle \rightarrow \langle e_1, s\rangle$.

7. $\langle$IF FALSE THEN e_1 ELSE $e_2, s\rangle \rightarrow \langle e_2, s\rangle$.

8. $\langle ref(i)[(v_2) := v_3], s\rangle \rightarrow \langle ref(m), s'\rangle$, where

   (a) $s \equiv [s[0], \ldots, s[i], \ldots, s[m-1]]$,

   (b) $s' = [s[0], \ldots, s[i], \ldots, s[m]]$,

   (c) $s[i] \equiv \#(w_0, \ldots, w_{n-1})$,

   (d) $v_2 \equiv j$, and

   (e) $s[m] = \#(w_0, \ldots, v_3, w_{j+1}, \ldots, w_{n-1})$.

9. $\langle ref(i)[(v_2) \leftarrow v_3], s\rangle \rightarrow \langle ref(i), s'\rangle$, where $s_1 \equiv [s[0], \ldots, s[i], \ldots, s[m-1]]$ and $s_2 \equiv [s[0], \ldots, s[i]\{v_2 \leftarrow v_3\}, \ldots, s[m-1]])$.

A *step* transforms a pair $\langle e, s\rangle$ consisting of a closed expression and a store, and is represented as $\langle e_1, s_1\rangle \longrightarrow \langle e_2, s_2\rangle$. A step $\langle E[r_1], s_1\rangle \longrightarrow \langle E[r_2], s_2\rangle$ holds if $\langle r_1, s_1\rangle \rightarrow \langle r_2, s_2\rangle$. The reflexive-transitive closure of $\longrightarrow$ is represented as $\langle e, s\rangle \xrightarrow{*} \langle e', s'\rangle$. If $\langle e, s\rangle \xrightarrow{*} \langle v, s'\rangle$, then the result of the computation is $s'(v)$, i.e., the result of replacing each reference $ref(i)$ in v by $s'[i]$. The computation of a closed term e is initiated on an empty store as $\langle e, []\rangle$. The value $reval(e)$ is defined to be $s(v)$, where $\langle e, []\rangle \xrightarrow{*} \langle v, s\rangle$.

A step in the destructive calculus is represented as $\langle e_1, s_1\rangle \Longrightarrow \langle e_2, s_2\rangle$ where $\langle E[r_1], s_1\rangle \Longrightarrow \langle E[r_2], s_2\rangle$ if $\langle r_1, s_1\rangle \Rightarrow \langle r_2, s_2\rangle$. As with $\longrightarrow$, we let $\stackrel{*}{\Longrightarrow}$ represent the reflexive-transitive closure of $\Longrightarrow$. The value $deval(e)$ is defined to be $s(v)$, where $\langle e, []\rangle \stackrel{*}{\Longrightarrow} \langle v, s\rangle$.

4 Observations

The correctness proof for the destructive update optimization is outlined in the Appendix. The proof demonstrates the existence of a bisimulation between evaluations of the unoptimized nondestructive program and the optimized program. The key idea in the proof is that the destructive optimizations always occur safely within update contexts during evaluation. When an update context coincides with an evaluation context, then the references accessible in the context are precisely those that govern the hole in the update context. The conditions on the occurrences of destructive operations within an update context then ensure that a reference that is updated destructively does not occur in the context. The operation $\gamma(a)$ transforms all destructive updates in a to corresponding nondestructive updates, and all destructive applications to the corresponding nondestructive applications. The observation that all destructive operations occur safely within update contexts can be used to construct a bisimulation between a nondestructive configuration $\langle e, s\rangle$ and a destructive configuration $\langle e', s'\rangle$ that holds when $s(e) = \gamma(s'(e'))$. The bisimulation easily yields the main theorem

$$reval(\sigma(e)) = deval(\sigma(e^D))$$

for closed e and environment σ that binds variables in e and e^D to values.

As already mentioned, our analysis is essentially similar to one given independently and earlier by Wand and Clinger [13], but our presentation and correctness proof are substantially simpler. The simplification in the proof derives from the use of evaluation contexts. Wand and Clinger use a configuration consisting of a label, an environment, a return context, and a continuation.

The worst-case complexity of analyzing a definition $f(x_1, \ldots, x_n) = e$ as described here is $n^2|e|$. The procedure requires n^2 iterations in the fixed point computation since $Lv(f)$ is an n-element array consisting of at most $n-1$ variables. Each iteration is linear in the size of the definition e. In practice, the complexity is much smaller since only a few variables are mutable.

We have used a simple core language for presenting the ideas. The method can be adapted to richer languages, but this has to be done carefully. For example, nested array structures introduce the possibility of structure sharing within an array that escapes the above analysis. For example, the update of index 3 of the array at index 2 of A might have the unintended side-effect of updating a shared reference at index 1 of array A. The analysis has to be extended to rule out the nested update of nested array structures. Non-nested updates of nested arrays such as $A(2)[(3) := 4]$ are already handled correctly by the analysis since the result is the updated inner array $A(2)$ and not the nested array A. Other nested structures such as records and tuples also admit similar structure sharing, but type information could be used to detect the absence of sharing.

Allowing array elements to be functional is only mildly problematic. Here, it is possible for references to the original array to be trapped in a function value (closure) as in $A[(2) := (\lambda(x) : x + A(2)(2))]$. It is easy to modify the notion of an update context and the accompanying definitions to handle functional values in arrays.

The analysis method can be adapted to lazy functional languages. Here, an additional analysis is needed to determine for a function $f(x_1, \ldots, x_n) = e$ if an argument x_j might be evaluated after an argument x_i in the body e of f.

PVS functions are translated to destructive Common Lisp operations that are then compiled and executed.[5] The typical performance of simple destructively optimized functional programs is within a linear factor of 5 of the corresponding C program in time, and with the same space behavior. For example, a "tiny" processor model from Rockwell Collins simulates roughly 1.25 million machine instructions per second using the PVS ground evaluator, which is about five times slower than a hand-built C-code simulator for the same processor model. (Using an optimizing compiler, the C-code can be boosted to nearly 10 million instructions per second.)

Conclusions. The mathematical orderliness of functional programming makes it possible to write efficient and easily optimizable programs. Optimizations arising from the static analysis for destructive update presented in this paper make it possible to execute functional programs with efficiency comparable to low-level imperative programs. The code generated by the PVS ground evaluator is so efficient that it is feasible to consider PVS as a programming language for applications such as safety-critical systems, where it is crucially important to run verified programs, and processor verification, where it is highly desirable to use a common specification for both verification and simulation.

[5] We are currently implementing a similar translator from PVS to Ocaml.

References

[1] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic '93.* pages 1–17, 1993.

[2] A. Bloss and P. Hudak. Path semantics. In *Proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics.* pages 476–489, 1987.

[3] Adrienne Bloss. Path analysis and the optimization of nonstrict functional languages. *ACM Transactions on Programming Languages and Systems*, 16(3):328–369, 1994.

[4] M. Draghicescu and S. Purushothaman. A uniform treatment of order of evaluation and aggregate update. *Theoretical Computer Science*, 118(2):231–262, September 1993.

[5] M. Felleisen. On the expressive power of programming languages. In *European Symposium on Programming.* pages 35–75, 1990.

[6] K. Gopinath and John L. Hennessy. Copy elimination in functional languages. In *16th ACM Symposium on Principles of Programming Languages.* January 1989.

[7] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis Horwood Ltd., 1987. Preliminary version appeared in Proceedings of 1986 ACM Conference on LISP and Functional Programming, August 1986, pages 351–363.

[8] Martin Odersky. How to make destructive updates less destructive. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 25–26, January 1991.

[9] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[10] John H. G. van Groningen. The implementation and efficiency of arrays in Clean 1.1. In *Proc. 8th International Workshop on Implementation of Functional Languages, IFL'96.* pages 105–124, 1996.

[11] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods.* North-Holland, Amsterdam, 1990.

[12] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

[13] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98.* pages 184–193, IEEE, April 1998.

A Correctness of Destructive Update Optimization

The correctness of the destructive update optimization is established by showing that the evaluations of the optimized and the unoptimized programs yield the same result. This result follows from the existence of a bisimulation between evaluations of the unoptimized nondestructive program and the optimized program. The key idea in the proof is that the destructive optimizations always occur *safely* within update contexts during evaluation. When a destructive subexpression is reduced during the evaluation, it occurs safely within the expression being evaluated, and this ensures that the bisimulation relation is preserved following the reduction.

The operation $\gamma(a)$ transforms all destructive updates $e_1[(e_2) \leftarrow e_3]$ in a to corresponding nondestructive updates $e_1[(e_2) := e_3]$, and all destructive applications $g^D(e_1, \ldots, e_n)$ to the corresponding nondestructive applications $g(e_1, \ldots, e_n)$.

An *environment* σ maps variables to values. The application of the substitution given by an environment σ to an expression e is written as $\sigma(e)$. Similarly, the application of a store s of the form $[s[0], \ldots, s[n-1]]]$ to an expression e, written as $s(e)$, replaces each occurrence of $ref(i)$ in e by $s[i]$, for $0 \leq i < n$.

The definition of safety for an expression e has been given in page 212. A destructive expression is either a destructive update of the form $e_1[e_2 \leftarrow e_3]$ or a destructive function invocation of the form $g^D(a_1, \ldots, a_n)$. An expression is safe if all destructive updates occur safely within update contexts. In the update analysis, the expressions being analyzed contained variables but no references. For the proof, the definitions of Ovr, Mv, and Av have to be extended to include references so that

$$Ovr(ref(i)) = Mv(ref(i)) = Av(ref(i)) = \{ref(i)\}.$$

The demonstration that all destructive operations occur safely within update contexts can be used to construct a bisimulation between a nondestructive configuration $\langle e, s\rangle$ and a destructive configuration $\langle e', s'\rangle$ that holds when $s(e) = \gamma(s'(e'))$. The bisimulation easily yields the main theorem

$$reval(\sigma(e)) = deval(\sigma(e^D)),$$

where $\sigma(e)$ and $\sigma(e^D)$ are closed terms.

The main observation is that the safety of the expression being evaluated is preserved by an evaluation step. All the terms considered in evaluations are closed. Let a safe term be one in which every destructive term occurs safely within an update context. In other words, e is safe if every occurrence of a destructive u of the form $e_1[e_2 \leftarrow e_3]$ or $g^D(a_1, \ldots, a_n)$ is safe in U^L, where $e \equiv U^L\{u\}$. An evaluation step always takes a safe closed term to a safe closed term.

The observation can be established by showing that whenever a safe term e goes to e' in an evaluation step $\langle e, s\rangle \Longrightarrow \langle e', s'\rangle$, then e' is also safe. We outline the argument before diving into the details of the proof. We need to show that each destructive term u in e' occurs safely. We can show that each destructive term in u in e' occurs within an update context V^M such that $e' \equiv V^M\{u\}$. In the evaluation step $\langle e, s\rangle \Longrightarrow \langle e', s'\rangle$, we reduce a redex r such that $\langle r, s\rangle \Rightarrow \langle r', s'\rangle$ where $e \equiv E[r]$ and $e' \equiv E[r']$ for an evaluation context E. It is an important property of redexes that the residue r' can either occur properly within u', coincide with u', properly contain u, or be independent of u'. That is, it is not possible for r' to partially overlap u'.

Let us call a term e *normal* if every occurrence of a destructive term u in e occurs within an update context, i.e., there is some update context U^L such that $e \equiv U^L\{u\}$. Normalcy is preserved during

evaluation: if e is a normal term and $\langle e, s\rangle \Longrightarrow \langle e', s'\rangle$, then so is e'. Essentially, U^L is an update context if the hole $\{\}$ in U does not occur within a lambda-abstraction. It is easy to see that none of the reductions cause a destructive term to appear within a lambda-abstraction. It is also easy to see that any evaluation context E can also be viewed as an update context of the form $U\{[]\}$, but not necessarily vice versa. Having shown that normalcy is preserved by evaluation, we can restrict our attention to the preservation of the safety conditions from page 212.

Let $dom(s)$ for a store s be the set $\{ref(i)|i < |s|\}$. A configuration $\langle e, s\rangle$ is called *well-formed* if each reference $ref(i)$ occurring in e is in the domain of s, $dom(s)$. The well-formedness of configurations is preserved during evaluation. The expression $s(e)$ contains no references when $\langle s, e\rangle$ is a well-formed configuration.

For every reduction $\langle r, s\rangle \Rightarrow \langle r', s'\rangle$, $\rho(r') \cap dom(s) \subseteq \rho(r)$. Hence, for every evaluation step $\langle e, s\rangle \Rightarrow \langle e', s'\rangle$, $\rho(e') \cap dom(s) \subseteq \rho(e)$.

For the preservation of safety, we have to show that every occurrence of a destructive term u' in an update context V^M in e' where $e' \equiv V^M\{u'\}$ is safe. Let e be of the form $E[r]$ and e' be of the form $E[r']$, where $\langle r, s\rangle \Rightarrow \langle r', s'\rangle$. Then, either r' occurs properly in u', u' occurs properly in r', or $r' \equiv u'$, or r' and u' do not overlap. This is because a redex cannot partially overlap a destructive term.

If r' occurs properly in u', then $e \equiv V^M\{u\}$, and the following cases arise:

1. u' is of the form $e_1'[e_2' \leftarrow e_3']$: Then, if r' occurs in either e_2' or e_3', we have that $e_1' \equiv e_1$, where $u \equiv e_1[e_2 \leftarrow e_3]$. Therefore $Ov(e_1') = Ov(e_1)$ and u' occurs safely within the update context V^M since u occurs safely within V^M.

2. u' is of the form $g^D(a_1', \ldots, a_n')$, where r' occurs in a_i for some i, $1 \leq i \leq n$. Then, $u \equiv g^D(a_1, \ldots, a_n)$, where $a_j \equiv a_j'$ for j, $1 \leq j \leq n$ and $i \neq j$. Since, $\langle e, s\rangle$ is a well-formed configuration, $M \subseteq dom(s)$. We also have that $Ov(a_i') \cap dom(s) \subseteq Ov(a_i)$, and $Av(a_i') \cap dom(s) \subseteq Av(a_i)$. Since $g^D(a_1, \ldots, a_n)$ occurs safely in V^M and $M \subseteq dom(s)$, it is also the case that $g^D(a_1', \ldots, a_n')$ occurs safely in the update context V^M. Thus, the safety of $\langle e', s'\rangle$ follows from that of $\langle e, s\rangle$.

If u' occurs (properly or not) within r', then by the syntax of a redex r, one of the following two cases is possible

1. r is a conditional expression and r' must be either the `THEN` or `ELSE` part of r. Either way, u' occurs safely in V^M since $e \equiv U^M\{u'\}$ and e is safe.

2. The reduction rule 2 has been applied in this step, a redex r of the form $g^D(v_1, \ldots, v_n)$ is reduced to r' of the form $\sigma(e^D)$, where $\sigma \equiv [v_1/x_1, \ldots, v_n/x_n]$. We have to ensure that u', which does not occur in e, is safe within the newly introduced update contexts. During the generation of e^D, we have already ensured that any destructive updates occur safely within update contexts. Thus, if u' is of the form $\sigma(e_1[(e_2) \leftarrow e_3])$, we have $e^D \equiv W^N\{e_1[(e_2) \leftarrow e_3]\}$ where $Ov(e_1) \cap N = \emptyset$. Since e^D does not contain any references, the result of applying environment σ to e^D is just $W'^{N'}\{\sigma(e_1)[\sigma(e_2) \leftarrow \sigma(e_3)]\}$, where $W' \equiv \sigma(W)$ and $N' \equiv \rho(\sigma(N))$. Note that for a value v, $Ov(v) \subseteq Av(v) \subseteq Mv(v) = \rho(v)$. From the update analysis of g, we know that $Ov(e_1) \cap N = \emptyset$ and $N \subseteq Lv(g)(x)$ for $x \in Ov(e_1)$. Note that $Ov(\sigma(e_1)) \subseteq \rho(\sigma(Ov(e_1)))$. Since the occurrence of $g^D(v_1, \ldots, v_n)$ is safe with respect to the update context U^L, we have that

(a) $Ov(\sigma(e_1)) \cap N' = \emptyset$

(b) $\rho(\sigma(Ov(e_1))) \cap L = \emptyset$, hence $Ov(\sigma(e_1)) \cap L = \emptyset$.

Therefore, the destructive update $\sigma(e_1)[\sigma(e_2) \leftarrow \sigma(e_3)]$ is safe with respect to the update context V^M since $V^M \equiv U^L\{W'^{N'}\}$, where $M = L \cup N'$.

A similar argument can be used to show that if u' is of the form $f^D(a_1, \ldots, a_n)$ in e^D, it occurs safely within the update context V^M.

Finally, if u' and r' do not overlap, then r' occurs in V^M and $\langle U^L, s\rangle \Longrightarrow \langle V^M, s'\rangle$. We can check that if U^L is of the form $E'[r]$ and V^M is $E'[r']$, then $M \subseteq (L - Av(r)) \cup Av(r')$. We can also check that for each reduction $\langle r, s\rangle \Rightarrow \langle r', s'\rangle$, $Av(r') \cap dom(s) \subseteq Av(r)$. Hence, $M \cap dom(s) \subseteq L$. Then obviously, u' occurs safely in V^M if it occurs safely in U^L.

This concludes the proof of the main invariant. The significance of the invariant should be obvious. It shows that whenever a destructive update redex $ref(i)[v_2 \leftarrow v_3]$ is evaluated, it occurs within a context that is both an evaluation context $E[]$ and an update context U^L. It is easy to check that L contains all the references in $e[]$. Since the destructive update $ref(i)[v_2 \leftarrow v_3]$ is safe for the update context U^L, the reference $ref(i)$ does not occur in $E[]$ and hence can be safely executed destructively.

Given that all configurations are well formed and safe, it is easy to establish the bisimulation between destructive and nondestructive execution. The bisimulation R between a nondestructive configuration $\langle e, s\rangle$ and a destructive configuration $\langle e', s'\rangle$ is given by $s(e) \equiv \gamma(s'(e'))$. It is now a routine matter to check that $\langle e_1, s_1\rangle \longrightarrow \langle e_2, s_2\rangle$ and $\langle e_1', s_1'\rangle \Longrightarrow \langle e_2', s_2'\rangle$ and $R(\langle e_1, s_1\rangle, \langle e_1', s_1'\rangle)$, then $R(\langle e_2, s_2\rangle, \langle e_2', s_2'\rangle)$.

The main correctness theorem easily follows from the bisimulation proof.

Theorem A.1 *If e and e' are closed, reference-free terms such that $\gamma(e') \equiv e$, then*

$$reval(e) \equiv deval(e).$$

PVS Code Proofs: Benchmarking and Enhancements

David Greve Matthew Wilding
Rockwell Collins, Inc.
Advanced Technology Center
Cedar Rapids, IA 52498 USA

Abstract

One challenge in proving that a computing system maintains separation between applications is reasoning about the behavior of the operating system code that does the system scheduling. General-purpose theorem proving programs offer the potential for highly reliable computing system verification, but harnessing theorem provers for this kind of activity poses some substantial challenges. This report discusses work to improve PVS's ability to reason about code execution. Work to make more automatic some kinds of PVS code proofs is reported in [8], and some of the introductory material in this report is adapted from it. The benchmarking work and identification of optimization opportunities was accomplished primarily by Rockwell Collins, and the PVS optimizations were implemented and tested primarily by SRI. Most of the work reported here was accomplished in the Fall of 1997 and Spring of 1998.

1 Introduction

Formal proofs about computer systems are often very complex and hard to get right, and the social process that is usually counted on to certify mathematical proofs is ineffective because particular computer system designs are often proprietary and in any case not of general interest. Mechanical theorem provers can help overcome both of these problems with formal proof: proofs generated with computer programs can be easier to produce and more reliable.

PVS is a verification system for "specifying and verifying digital systems" [4, 5, 7]. It supports a specification language that is based on a simply typed higher-order logic, and provides a large number of prover commands that allow machine-checked reasoning about expressions in the logic. There is support for automating reasoning in PVS, namely a simple rewriting system and a facility for constructing new proof commands, although the emphasis in PVS is on building clear specifications and supporting user proof with domain-specific decision procedures.

Earlier work at Rockwell Collins and SRI verifying aspects of the Rockwell AAMP5 and AAMP-FV processor designs with microcoded instruction sets is reported in [2, 3]. Partial microcode correctness of these processors has been established using PVS. The hardware that executes microcode was formalized in the PVS logic, and proofs that the microcode correctly implements some of the processor instruction sets have been constructed. While the application of PVS to realistic-sized processors in the AAMP5 and AAMP-FV projects led to a partial verification of their microcode, the experience of building these proofs led the developers to the pragmatic realization that practical computer systems proofs must be robust [2]. That is, computer system proofs must be able to demonstrate correctness with minimal human assistance despite modest system or specification changes.

Mistakes in proof development and changes to system design and specification are inevitable for realistic-sized verifications. For example, during the AAMP-FV verification effort a change was made in the formal model related to memory address decoding [2]. This change caused every previously-constructed instruction correctness proof to fail even though the change had little to do with the substance of most of the proofs. Large programming projects use software engineering techniques to make software robust despite inevitable changes. So too must large machine-checked proof projects use techniques to develop robust proofs.

In the next section we present a formalization of the simple computing system and a benchmark, both previously introduced in [8]. In order to improve PVS code proof capabilities, we used this benchmark to illustrate scaling problems that were fixed. We then outline how we adapted the techniques of [8] that foster robustness to reason about AAMP-FV code. Rockwell Collins developed these benchmarks and uncovered some optimization opportunities in PVS, and SRI developed and implemented the improvements.

move a b store value at location b in location a
movei a n move value n in location a
movewind a b store value at location b in location stored at location a
moverind a b store value at the location stored at location b in location a
add a b store sum of values at locations a and b in location a
sub a b store in location a the greater of 0 and the difference of a and b
incr a increment value at location a
decr a decrement value at location a
jump n store value n in pc
jumpz a n store value n in pc if value at location a is 0.
call n store (incremented) pc on the stack and store value n in pc
ret store a value popped from the stack in pc
halt set the halt flag

Figure 1: The sm Instructions

2 A benchmark based on a simple machine interpreter

We initially worked on models of "small machine" or "sm" [8]. This toy computing system model is a slightly modified version of John Rushby's formalization of Bob Boyer's and J Moore's simple machine-level language [1, 6]. It is far simpler than realistic device models, but allows us to focus on fundamental issues related to code proofs before we add the complexity of a realistic model such as the AAMP-FV.

As introduced in [8], an sm state is composed of five elements: a program counter, a stack containing subroutine call return addresses, a data memory that maps natural number addresses to natural number values, a flag whose boolean value indicates whether the processor is halted, and a program memory that maps natural number addresses to instructions. Both instruction and data memory size at 100 elements which limits the valid addresses for the memories to values less than 100. Each sm instruction is a record containing one of 13 opcodes and two addresses. The instructions are described informally in Figure 2.

Following the style of some previous code proof efforts using other theorem proving systems [1, 9], we introduce an interpreter that provides an operationalaxiomatic specification of the execution of the machine. The function step defines precisely the effect of executing the instruction pointed to by the pc, thereby providing a formal version of the instruction descriptions of Figure 2 with which we can reason about programs. We define a function sm that returns the state resulting from running n instructions starting in state s.

```
sm(s: state, n: nat): RECURSIVE state =
  IF n = 0 THEN s ELSE sm(step(s), n - 1) ENDIF
 MEASURE n
```

address	code
0	move 2 0
1	move 3 0
2	move 4 1
3	sub 4 2
4	jumpz 4 12
5	incr 2
6	moverind 4 2
7	moverind 5 3
8	sub 5 4
9	jumpz 5 2
10	move 3 2
11	jump 2
12	ret

Figure 2: A benchmark program [8]

Figure 2 presents a "min" program that returns in register 3 the location of a least element of the array whose bounds are contained in registers 0 and 1. A discussion of the issues related to the specification and PVS proof of this program appears in [8]. The verification relies on PVS's builtin simplification procedures that among other things apply previously-proved theorems as rewrite rules. As a first step in benchmarking and improving PVS's ability to handle computing models of realistic size and complexity, we experimented with adding complexity to the simple program of Figure 2 in various simple ways. This approach helped us identify aspects of PVS that have the potential to cause problems on real computational models. By initially starting with a simple computational model we simplified the later optimization process.

One of the approaches that identified a promising PVS optimization was to to add effective "no-op" statements to the program's loop and to measure the effect on the time required to process the proof. Ideally one would expect that the time required to complete a proof that requires symbolic execution of code would require time approximately linear in the number of instructions at issue. When the symbolic state of a machine is updated with the effect of a single instruction, it should be possible to calculate the effect of the next instruction in roughly the time that would have been required had the first instruction's effect not been calculated.

Figure 3 shows the time required to execute the benchmark program loop once through symbolically on the PVS available before the optimizations resulting from this program.[1]. The X-axis indicates how many effective "no-ops" were added to the code, and the y-axis indicates how long the proof requires in seconds.

Trial 1 uses perhaps the most straightforward set of simplifiers, that has the effect during simplification of opening the interpreter `sm` until only a nest of

[1] on an unloaded Sparcstation 5 running PVS version 2.1 Test (patch level 2.399)

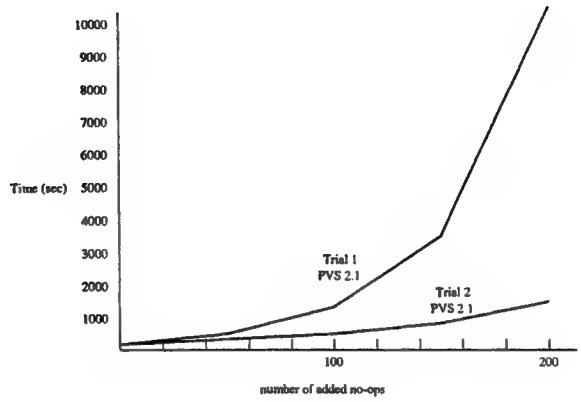


Figure 3: Timing of simple PVS code correctness proof

`step` remains, then applying the definition of `step` until the symbolic expression representing the result remains. Trial 2 improves on this approach somewhat, by opening the interpreter and simplifying the step function definition immediately. Generation of the large term of `step` functions slowed proofs in PVS, which we noted for future PVS proofs.

Note that neither trial achieved the linear timing for which we hope. Although trial 2 is "close", note that `sm` is very simple, and far less complex than the models we actually care about involving real computing systems. The nonlinearity was basically caused by PVS 2.1's elegant formalization of record structures, which treats them essentially as functions. An unfortunate practical implication of this approach was the flexibility with which the user could adjust the domain - and therefore the type! - of functions. This capability is not necessary for the typical use of records, but its treatment in this way led to unnecessary inefficiency in the PVS simplifier.

SRI implemented several optimizations in response to these problems. Most visibly, record updates that do not change the domain now are distinguished from updates that may change the domain in the logic. This allows more efficient manipulation for typical use.

3 A Simple Benchmark Based on a Real Machine Model

In the previous section we described our use of a toy computing model to identify PVS optimization opportunities. Once we exhausted the improvements we could make with the toy, the next step was to develop a realistic model for use on the project. We describe in this section this model, and our benchmark use of it.

The AAMP-FV is a paper-and-pencil design of a realistic flight control pro-

cessor. An earlier project pursued the formal specification and verification of the AAMP-FV using the PVS theorem prover [2]. This work related the execution of microcode to the formal specification of the instructions for the processor, and did so for many of the instructions using the capabilities of the PVS theorem prover. We adapted the instruction-level model of the AAMP-FV.

The adaptation of the model for this project required several changes from that developed in [2] in order to make it consistent with the robust proof approach from [8]. In particular,

- Integers are used to represent the bit-vectors of the machine state, rather than a function mapping bit-vector location to bits.
- A PVS record is used to represent the state, rather than a list of axiomatized functions describing state elements.
- An explicit interpreter and step function definitions similar in style to `sm` is introduced to describe the model, rather than rely on the PVS "axiom" feature.

In order to reason about AAMP-FV code, a toolchain that produced PVS-readable object code from AAMP-FV assembly language was developed. This was relatively straightforward: PVS rewrite rules were developed that "read" from an axiomatized variable, and axioms representing the bytes of object code were introduced. The axioms are generated by a short Lisp program that reads a standard AAMP-FV object file. A preexisting assembler generates this object file.

A benchmark program we call "dummkopf" allowed testing of the code. Fragments of this code are listed in Figure 4. The dummkopf code flips between user and executive mode, incrementing two counters counters that keep track of the number of flips.

Initial application of PVS to dummkopf was disappointing. After a few instructions are symbolically executed, the prover slowed to a crawl as a result. It was not possible to reason about the execution of more than about 10 instructions at a time. However, SRI implemented a number of optimizations that made possible symbolic simulation of arbitrarily long sequences of AAMP-FV code. Current versions of PVS now support symbolic simulation in times approximately linear with the number of instructions.

4 Summary

A crucial capability for proving security or safety properties of computing system is reasoning about the execution of software. Computing platform models and benchmark code identified several ways in which PVS could be improved to support reasoning about code execution and optimizations implemented by SRI significantly improved PVS in this area.

```
ELOOP:  LIT16   OFFFFh                  ;
        ASN24   EXEC_CODE               ;
        LIT24   PSD_U0                  ;
        USER                            ; Activate user task
        ASN24   EXEC_CODE               ; Write out TRAP code
        REF24   COUNT_E                 ; Increment COUNT_E
        LIT4    1                       ;
        ADD                             ;
        ASN24   COUNT_E                 ;
        LIT16   ELOOP.b. - $.b. - 4     ;
        SKIP                            ; Loop
...

SUB_0: .dw      08000h | 0              ;
        .dd     PAGE_U0                 ;
        LIT4    0                       ; Clear iteration counter (COUNT_U0)
        ASN24   COUNT_U0                ;
LOOP_0: REF24   COUNT_U0                ; Increment COUNT_U0
        LIT4    1                       ;
        ADD                             ;
        ASN24   COUNT_U0                ;
        TRAP                            ; Trap to executive
        LIT16   LOOP_0.b. - $.b. - 4    ;
        SKIP                            ; Loop
; Should never get here ;-)
        LIT4    0                       ; deallocation amount
        RETURN                          ; return to primal proc.
```

Figure 4: Fragments of Dummkopf, an AAMP-FV assembly benchmark

References

[1] Robert S. Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1996.

[2] Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.

[3] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.

[4] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[5] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[6] John Rushby. Personal Communication, November 1996.

[7] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[8] Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356, 1997. (http://atb-www.larc.nasa.gov/Lfm97/).

[9] Phillip J. Windley. A theory of generic interpreters. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

Invited paper presented at the second IEEE International Conference on Formal Engineering Methods (ICFEM '98), pp. 176–178, Brisbane, Australia, December, 1998.

Ubiquitous Abstraction: A New Approach for Mechanized Formal Verification*

Extended Abstract

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

*This work was supported by Darpa through USAF Rome Laboratory Contract No. F30602-96-C-0204, and by the National Science Foundation under contract CCR-9509931.

Formal methods can provide many benefits but, to my mind, the chief benefit of specifically *formal* methods is that they allow some properties of a computational system to be deduced from its design by a process of *logical calculation*, in much the same way that computational fluid dynamics allow properties of aerofoils to be examined by numerical calculation.

Originally, "computational system" meant *computer program*, and the main property of interest was correctness of the program with respect to its specification. More recently, however, these notions have widened to include almost any level of system description (e.g., hardware, algorithms, software architecture, requirements), properties short of full "correctness" (e.g., various notions of internal and external consistency), and refutation (i.e., bug finding) as much as verification. Mechanized formal verification uses the techniques of automated deduction—that is theorem proving and model checking—to perform the "logical calculations" that enable such properties to be checked for such system descriptions. The most successful verification systems combine an interactive theorem prover with powerful automation such as decision procedures for equality and arithmetic, and rewriting: the user directs the overall process, while the automation takes care of the details.

With the aid of a modern verification system, routine formal analyses are, well, routine. By this I mean that if the property of interest follows fairly directly from the system description by reasoning in some previously formalized mathematical domains, then mechanized formal verification is unlikely to be more difficult or to take longer—and may be considerably easier, as well as less error-prone—than a comparably detailed informal examination. Much worthwhile analysis can be accomplished economically and reliably in this way (see, for example, [4], which describes analysis of tables and other requirements specifications for some recent Space Shuttle software), but there is much else that can be accomplished only with great difficulty and effort.

These more challenging problems often involve concurrency, as in protocols and distributed algorithms, and the difficulties are not so much in theorem proving as in ancillary tasks, such as the invention of suitably strong invariants, and diagnosing whether an intractable subgoal indicates an error in the design, an inadequate invariant, or a mistaken proof step. To establish that a concurrent system (typically specified as a transition relation) maintains a desired invariant (expressing some safety property, for example), the basic deductive method is to show that the invariant is implied by the initial system state(s), and that it is preserved by all transitions. Usually, the desired property is *not* preserved in this simple manner, and it is necessary to strengthen it with additional conjuncts to characterize the reachable states (since preservation is required only for states that can be reached from the initial states). These conjuncts can often be found—one at a time—by inspecting a failed proof, extracting a plausible conjunct, and repeating the process until the proof succeeds. In one well-known example, 57 iterations of this kind were required to verify a relatively simple communications protocol known as the "bounded retransmission protocol" [5].

Model checking is an attractive alternative to theorem proving in circumstances such as these. Model checking is largely automatic, but it is applicable only to finite state systems (and to some infinite state systems having special forms); consequently, most system descriptions must be "downscaled" (i.e., aggressively simplified) before they can be subjected to model checking. Unless there is a suitable abstraction (i.e., simulation) relationship between the original system description and the downscaled one, model checking may be unsound or incomplete with respect to the original system: that is, it may fail to detect an error (because it is not present in the downscaled system), or may falsely report errors (that are present in the downscaled system but not in the original). The latter is not much of a problem when refuation is the goal: model checkers generally produce a counterexample in the form of an execution trace that manifests the error in the abstracted system, and it is usually straightforward to check whether a corresponding trace leads to an error in the original system. This may be adequate for refutation, but for verification we need to know that the model checker's inability to find errors in the downscaled system implies satisfaction of the desired property by the original system. For this, it is necessary to establish a suitable abstraction relationship between the original and the downscaled system descriptions—and doing so by traditional means can be almost as hard as proving the property directly. For the example of the bounded retransmission protocol, justification of an abstraction for model checking required 45 of the 57 conjuncts used in the direct proof.

Recently, several researchers, including my colleagues at SRI, have been exploring ways to combine theorem proving, abstraction, model checking, and other techniques more aggressively than before in order to avoid some of the difficulties and costs described above.

One idea is to *calculate* an abstracted system description, so that it is correct by construction, rather than to justify a downscaled description constructed by hand. Given an abstraction function relating the original and abstracted state spaces, a verification condition can be generated for each pair of abstract states that specifies the conditions under which no transition is required between those states in the abstracted system description (the condition is that there is no transition between any pair of original states that map to those abstract states). If the verification condition can be proved (using automatic proof procedures), then the transition can be omitted from the abstracted system description; if not, then it is conservative to include the transition. For the bounded retransmission protocol, this approach is able to compute automatically an abstracted system description suitable for model checking [1]. More sophisticated treatments allow the desired invariant to be used in construction of the abstracted system, and can use information from a failed model check to refine the abstraction.

Calculation of abstracted system descriptions often requires, and is usually made easier, if known invariants can be supplied to the process. Some useful invariants can be calculated by static analysis [3], but others can be extracted from model checking. As part of its computation, a model checker will almost certainly calculate the set of reachable states of the system description presented to it. Now, the reachable states of a system characterize its strongest invariant, so a concretization of the reachable states of an abstracted system is certainly an invariant, and possibly a strong one, for the original system.[1] This suggests a new way to calculate invariants that may help in the construction of abstracted system descriptions: construct some simpler abstraction (one for which already known invariants are adequate for its construction), and use a concretization of its reachable states as a new invariant. A practical difficulty in this approach is that the reachable state set calculated by a model checker is not usually made available externally and, in any case, it is usually represented by a data structure (a BDD) that is not directly suitable for input to a theorem prover. This difficulty has been overcome in the current version of the SMV model checker, where a `print` function, implemented by Sergey Berezin, provides external access to the reachable states.

The techniques described so far allow calculation of invariants and of abstracted systems, but they require the user to supply suitable abstraction functions. Some guidance in doing this can be obtained by inspecting the predicates that appear in the original, concrete, system description (particularly those in the guards on transitions): if a predicate such as $x = y + 1 \wedge x \neq z$ appears in the concrete system description, then an abstraction can be constructed having a boolean state variable that records the truth or falsity of this predicate [8]. Even with the aid of heuristics such as this, however, it can still require great insight to design a tractable abstraction that preserves the property of interest.

An alternative approach does not seek to construct an abstraction that directly preserves the property of interest: instead, this approach uses theorem proving as its top-level technique, and employs abstraction and model checking to help discharge the subgoals that are generated [7]. The attraction here is that theorem proving will have performed some case analysis in generating the subgoals, so that they will be simpler than the original problem. Therefore the abstraction needed to help discharge a given subgoal can be much simpler than one that discharges the whole problem; furthermore the predicates that appear in the formulas of the subgoal provide useful hints for the construction of a suitable abstraction.

In summary, "ubiquitous abstraction"—that is constructing many different abstracted system descriptions at many different points in an analysis, and for several different purposes—has great promise as a way to ease difficulties and increase productivity and automation in the formal analysis of concurrent systems. The approach also provides a new way to combine different tools, such as theorem provers and model checkers, though full exploitation of this opportunity requires modification to the tools so that they can exchange symbolic values (e.g., the reachable state set, or a counterexample) rather than merely report the success or failure

[1] "Concretization" is the inverse of abstraction; the inverse of the abstraction function is not a function, in general, so some approximation is required to find a set of concrete states who image under the abstraction function includes all the reachable abstract states.

of their own local analysis. Some of the capabilities I have described are already integrated in a system called InVeSt [2] and initial experiments with this and other prototypes developed as part of our "Symbolic Analysis Laboratory" (SAL) are quite promising. Our current plans are to evaluate the approach on more challenging examples.

Acknowledgements

The ideas outlined here, and the systems that implement them, are the work of my colleagues Saddek Bensalem, Sergey Berezin, Yassine Lakhnech, Sam Owre, Hassen Saïdi, and Natarajan Shankar.

References

Papers by SRI authors can generally be found at `http://www.csl.sri.com/fm.html`.

[1] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [6], pages 319–331.

[2] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [6], pages 505–510.

[3] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[4] J. Crow and B. L. Di Vito. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

[5] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, Mar. 1996. Springer-Verlag.

[6] A. J. Hu and M. Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[7] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 178–192, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.

[8] H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

In C. Michael Holloway, editor, LFM 2000: Fifth NASA Langley Formal Methods Workshop, Williamsburg, VA, June 2000. Proceedings available at http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/

An Overview of SAL*

Saddek Bensalem[†] Vijay Ganesh[‡] Yassine Lakhnech[†] Cesar Muñoz[§]
Sam Owre[¶] Harald Rueß[¶] John Rushby[¶] Vlad Rusu[‖] Hassen Saïdi[**]
N. Shankar[¶] Eli Singerman[††] Ashish Tiwari[‡‡]

*This research was performed in the Computer Science Laboratory, SRI International, Menlo Park CA USA, and supported by DARPA through USAF Rome Laboratory contract F30602-96-C-0204, by NASA Langley Research Center contract NAS1-20334, and by the National Science Foundation contract CCR-9509931.

[†]VERIMAG, Grenoble, France
[‡]Stanford University, Stanford CA
[§]ICASE, NASA Langley, Hampton VA
[¶]Computer Science Laboratory, SRI International, Menlo Park CA
[‖]IRISA, Rennes, France
[**]Systems Design Laboratory, SRI International, Menlo Park CA
[††]Intel, Haifa, Israel
[‡‡]SUNY Stony Brook, NY

Abstract

To become practical for assurance, automated formal methods must be made more scalable, automatic, and cost-effective. Such an increase in scope, scale, automation, and utility can be derived from an emphasis on a systematic separation of concerns during verification. SAL (Symbolic Analysis Laboratory) attempts to address these issues. It is a framework for combining different tools to calculate properties of concurrent systems. The heart of SAL is a language, developed in collaboration with Stanford, Berkeley, and Verimag, for specifying concurrent systems in a compositional way. Our instantiation of the SAL framework augments PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking to separate concerns as well as calculate properties (i.e., perform symbolic analysis) of concurrent systems. We describe the motivation, the language, the tools, their integration in SAL/PVS, and some preliminary experience of their use.

1 Introduction

To become practical for debugging, assurance, and certification, formal methods must be made more cost-effective. Incremental improvements to individual verification techniques will not suffice. It is our basic premise that a significant advance in the effectiveness and automation of verification of concurrent systems is possible by engineering a systematic separation of concerns through a truly integrated combination of static analysis, model checking, and theorem proving techniques. A key idea is to change the perception (and implementation) of model checkers and theorem provers from tools that perform verifications to ones that calculate *properties* such as slices, abstractions and invariants. In this way, big problems are cut down to manageable size, and properties of big systems emerge from those of reduced subsystems obtained by slicing, abstraction, and composition. By iterating through several such steps, it becomes possible to incrementally accumulate properties that eventually enable computation of a substantial new property—which in turn enables accumulation of further properties. By interacting at the level of properties and abstractions, multiple analysis tools can be used to derive properties that are beyond the capabilities of any individual tool.

SAL (Symbolic Analysis Laboratory) addresses these issues. It is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking toward the calculation of properties (symbolic analysis) of concurrent systems expressed as transition systems. The heart of SAL is an intermediate language, developed in collaboration with Stanford, Berkeley, and Verimag for specifying concurrent systems in a compositional way. This language will serve as the target for translators that extract the transition system description for popular programming languages such as Esterel, Java, or Verilog. The intermediate language also serves as a common description from which different analysis tools can be driven by translating the intermediate language to the input format for the tools and translating the output of these tools back to the SAL intermediate language.

This paper is structured as follows. In Section 2 we describe the motivation and rationale behind the design of the SAL language and give an overview of its main features. The main part, Section 3, describes SAL components including slicing, invariant generation, abstraction, model checking, simulation, and theorem proving together with their integration into the SAL toolset. Section 4 concludes with some remarks.

2 The SAL Common Intermediate Language

Mechanized formal analysis starts from a description of the problem of interest expressed in the notation of the tool to be employed. Construction of this description often entails considerable work: first to recast the system specification from its native expression in C, Esterel, Java, SCR, UML, Verilog, or whatever, into the notation of the tool concerned, then to extract the part that is relevant to the analysis at hand, and finally to reduce it to a form that the tool can handle. If a second tool is to be employed for a different analysis, then a second description of the problem must be prepared, with considerable duplication of effort. With m source languages and n tools, we need $m*n$ translators. This situation naturally suggests use of a common intermediate language, where the numbers of tools required could be reduced to $m + n$ translators.

The intermediate language must serve as a medium for representing the state transition semantics of a system described in a source language such as Java or Esterel. It must also serve as a common representation for driving a number of back-end tools such as theorem provers and model checkers. A useful intermediate language for describing concurrent systems must attempt to preserve both the structure and meaning of the original specification while supporting a modular analysis of the transition system.

For these reasons, the SAL intermediate language is a rather rich language. In the sequel, we give an overview

```
mutex : CONTEXT =
BEGIN

PC: TYPE = {trying, criti-
cal, sleeping}

mutex [tval:boolean] : MODULE =
BEGIN
INPUT  pc2: PC, x2: boolean
OUTPUT pc1: PC, x1: boolean

INITIALIZATION
  TRUE -->  pc1 = sleeping;
            x1  = tval

TRANSITION
   pc1 = sleeping
      --> pc1' = trying;
           x1' = (x2=tval)
  []
   pc1 = trying AND
    (pc2=sleeping OR x1= (x2/=tval))
      --> pc1' = critical
  []
   pc1 = critical
      --> pc1' = sleeping;
           x1' = (x2=tval)
END

system: MODULE =
  HIDE x1,x2
   (mutex[FALSE]
    || RENAME pc2 TO pc1,
              x2 TO x1,
              pc1 TO pc2,
              x1 TO x2
         mutex[TRUE])

mutualExclusion: THEOREM
  system |-
     AG(NOT(pc1=critical
            AND pc2=critical))

eventually1: LEMMA
     system |- EF(pc1=critical)
eventually2: LEMMA
     system |- EF(pc2=critical)

END
```

Figure 1. Mutual Exclusion

of the main features of the SAL type language, the expression language, the module language, and the context language. For a precise definition and semantics of the SAL language, including comparisons to related languages for expressing concurrent systems, see [31].

The type system of SAL supports basic types such as booleans, scalars, integers and integer subranges, records, arrays, and abstract datatypes. Expressions are strongly typed. The expressions consist of constants, variables, applications of Boolean, arithmetic, and bit-vector operations (bit-vectors are just arrays of Booleans), and array and record selection and updates. Conditional expressions are also part of the expression language and user-defined functions may also be introduced.

A module is a self-contained specification of a transition system in SAL. Usually, several modules are collected in a context. Contexts also include type and constant declarations. A transition system *module* consists of a *state* type, an *initialization condition* on this state type, and a binary *transition relation* of a specific form on the state type. The state type is defined by four pairwise disjoint sets of *input*, *output*, *global*, and *local* variables. The input and global variables are the *observed* variables of a module and the output, global, and local variables are the *controlled* variables of the module. It is good pragmatics to name a module. This name can be used to index the local variables so that they need not be renamed during composition. Also, the properties of the module can be indexed on the name for quick lookup.

Consider, for example, the SAL specification of a variant of Peterson's mutual exclusion algorithm in Figure 1. Here the state of the module consists of the controlled variables corresponding to its own program counter `pc1` and boolean variable `x1`, and the observed variables are the corresponding `pc2` and `x2` of the other process.

The transitions of a module can be specified variable-wise by means of *definitions* or transition-wise by *guarded commands*. Henceforth, primed variables `X'` denote next-state variables. A definition is of the form `X = f(Y, Z)`. Both the initializations and transitions can also be specified as guarded assignments. Each guarded command consists of a guarded formula and an assignment part. The guard is a boolean expression in the current controlled (local, global, and output) variables and current-state and next-state input variables. The assignment part is a list of equalities between a left-hand side next-state variable and a right hand side expression in both current-state and next-state variables.

Parametric modules allow the use of logical (state-independent) and type parameterization in the definition of modules. Module `mutex` in Figure 1, for example, is

parametric in the Boolean `tval`. Furthermore, modules in SAL can be combined by either synchronous composition `||`, or asynchronous composition `[]`. Two instances of the `mutex` module, for example, are conjoined synchronously to form a module called `system` in Figure 1. This combination also uses *hiding* and *renaming*. Output and global variables can be made local by the `HIDE` construct. In order to avoid name clashes, variables in a module can be renamed using the `RENAME` construct.

Besides declaring new types, constants, or modules, SAL also includes constructs for stating module properties and abstractions between modules. CTL formulas are used, for example, in Figure 1 to state safety and liveness properties about the combined module `system`.

The form of composition in SAL supports a compositional analysis in the sense that any module properties expressed in linear-time temporal logic or in the more expressive universal fragment of CTL* are preserved through composition. A similar claim holds for asynchronous composition with respect to stuttering invariant properties where a stuttering step is one where the local and output variables of the module remain unchanged.

Because SAL is an environment where theorem proving as well as model checking is available, absence of causal loops in synchronous systems is ensured by generating proof obligations, rather than by more restrictive syntactic methods as in other languages. Consider the following definitions:

```
X = IF A THEN NOT Y ELSE C ENDIF
Y = IF A THEN B ELSE X ENDIF
```

This pair of definitions is acceptable in SAL because we can prove that `X` is *causally dependent* on `Y` only when `A` is *true*, and vice-versa only when it is *false*—hence there is no causal loop. In general, *causality checking* generates proof obligations asserting that the conditions that can trigger a causal loop are unreachable.

3 SAL Components

SAL is built around a blackboard architecture centered around the SAL intermediate language. Different backend tools operate on system descriptions in the intermediate language to generate properties and abstractions. The core of the SAL toolset includes the usual infrastructure for parsing and type-checking. It also allows integration of translators and specialized components for computing and verifying properties of transition systems. These components are loosely coupled and communicate through well-defined interfaces. An invariant generator may expect, for example, various application specific flags and a SAL base module, and it generates a corresponding assertion in the context language together with a justification of the invariant. The SAL toolset keeps track of the dependencies between generated entities, and provides capabilities similar to proof-chain analysis in theorem proving systems like PVS.

The main ingredients of the SAL toolset are specialized components for computing and verifying properties of transition systems. Currently, we have integrated various components providing basic capabilities for analyzing SAL specifications, including

- Validation based on theorem proving, model checking, and animation;
- Abstraction and invariant generation;
- Generation of counterexamples;
- Slicing.

We describe these components in more detail below.

3.1 Backend translations

We have developed translators from the SAL intermediate language to PVS, SMV, and Java for validating SAL specifications by means of theorem proving (in PVS), model checking (in SMV), and animation (in Java). These compilers implement *shallow structural embeddings* [26] of the SAL language; that is, SAL types and expressions are given a semantics with respect to a model defined by the logic of the target language. The compilers performs a limited set of semantic checks. These checks mainly concern the use of state variables. More complex checks, as for example type checking, are left to the verification tools.

3.1.1 Theorem Proving: SAL to PVS

PVS is a specification and verification environment based on higher-order logic [27]. SAL contexts containing definitions of types, constants, and modules, are translated into PVS theories. This translation yields a semantics for SAL transition systems. Modules are translated as parametric theories containing a record type to represent the state type, a predicate over states to represent the initialization condition, and a relation over states to represent the transition relation. Figure 2 describes a typical translation of a SAL module in PVS. Notice that initializations as well as transitions may be nondeterministic.

```
module[para:Parameters] : THEORY
BEGIN
  State : TYPE = [#
    input  : InputVars,
    output : OutputVars,
    local  : LocalVars
  #]

  state,next : VAR State

  initialization(state):boolean =
    (guard_init_1 AND
     output(state) = ... AND
     local(state) = ...)
    OR ... OR (guard_init_n AND ...)

  transitions(state, next):boolean =
    (guard_trans_1 AND
     output(next) =
          output(state) WITH [...]
     local(next) =
          local(state) WITH [... ])
    OR ... OR
    (guard_trans_m AND ...)
    OR
    (NOT guard_trans_1 AND ... AND
     NOT guard_trans_m AND
     output(next) = output(state)
     local(next) = local(state))
```

Figure 2. A SAL module in PVS

Compositions of modules are embedded as logical operations on the transition relations of the corresponding modules: disjunction for the case of asynchronous composition, conjunction for the case of synchronous composition. Hiding and renaming operations are modeled as morphisms on the state types of the modules. Logical properties are encoded via the temporal logic of the PVS specification language.

3.1.2 Model Checking: SAL to SMV

SMV is a popular model checker with its own system description language [25]. SAL modules are mapped to SMV modules. Type and constant definitions appearing in SAL contexts are directly expanded in the SMV specifications. Output and local variables are translated to variables in SMV. Input variables are encoded as parameters of SMV modules.

The nondeterministic assignment of SMV is used to capture the arbitrary choice of an enabled SAL transition. Roughly speaking, two extra variables are introduced. The first is assigned nondeterministically with a value representing a SAL transition. The guard of the transition represented by this variable is the first guard to be evaluated. The second variable loops over all transitions starting from the chosen one until it finds a transition which is enabled. This mechanism assures that every transition satisfying the guard has an equal chance to being fired in the first place. Composition of SAL modules and logical properties are directly translated via the specification language of SMV.

3.1.3 Animation: SAL to Java

Animation of SAL specifications is possible via compilation to Java. However, not all the features of the SAL language are supported by the compiler. In particular, the expression language that is supported is limited to that of Java. For example, only integers and booleans are accepted as basic types. Elements of enumeration types are translated as constants and record types are represented by classes.

The state type of a SAL module is represented by a class containing fields for the input, output, and local variables. In order to simulate the nondeterminism of the initialization conditions, we have implemented a random function that arbitrary chooses one of the initialization transition satisfying the guard.

Each transition is translated as a Java thread class. At execution time, all the threads share the same state object. We assume that the Java virtual Machine is nondeterministic with respect to execution of threads. The main function of the Java translation creates one state object and passes the object as an argument to the thread object constructors. It then starts all the threads. Safety properties are encoded by using the exception mechanism of Java, and are checked at run time.

3.1.4 Case Study: Flight Guidance System

Mode confusion is a concern in aviation safety. It occurs when pilots get confused about the actual states of the flight deck automation. NASA Langley conducts research to formally characterize mode confusion situations in avionics systems. In particular, a prototype of a Flight Guidance System (FGS) has been selected a case study for the application of formal techniques to identify mode confusion problems. FGS has been specified in various formalisms (see [23] for a comprehensive list of related work). Based on work by Lüttgen and Carreño, we have developed a complete specification of FGS in SAL. The specification has been automatically translated to SMV and PVS, where it has been analyzed. We did not experience any significant overhead in model checking translated SAL models compared to hand-coded SMV models. This case study is

available at `http://www.icase.edu./~munoz/sources.html`.

3.2 Invariant Generation

An *invariant* of a transition system is an assertion—a predicate on the state—that holds of every reachable state of the transition system. An *inductive invariant* is a assertion that holds of the initial states and is preserved by each transition of the transition system. An inductive invariant is also an invariant but not every invariant is inductive.

Let $\mathtt{SP}(\mathcal{T}, \phi)$ denote the formula that represents the set of all states that can be reached from any state in ϕ via a single transition of the system $\mathcal{T}$, and Θ denote the formula that denotes the initial states. A formula ϕ is an inductive invariant for the transition system $\mathcal{T}$ if (i) $\Theta \rightarrow \phi$; (ii) $\mathtt{SP}(\mathcal{T}, \phi) \rightarrow \phi$.

We recall that for a given transition system $\mathcal{T}$ and a set of states described by formula ϕ, the notation $\mathtt{SP}(\mathcal{T}, \phi)$ denotes the formula that characterizes all states reachable from states ϕ using exactly one transition from $\mathcal{T}$. If Θ denotes the initial state, then it follows from the definition of invariants that any fixed-point of the operator $F(\phi) = \mathtt{SP}(\mathcal{T}, \phi) \vee \Theta$ is an invariant.

Notice that the computation of strongest postconditions introduces existentially quantified formulas. Due to novel theorem proving techniques in PVS2.3 that are based on the combination of a set of ground decision procedures and quantifier elimination we are able to effectively reason about these formulas in many interesting cases.

It is a simple observation that not only is the greatest fixed point of the above operator an invariant, but every intermediate ϕ_i generated in an iterated computation procedure of greatest fixed point also is an invariant.

$$\begin{aligned} \phi_0 &: \mathtt{true} \\ \phi_{i+1} &: \mathtt{SP}(\mathcal{T}, \phi_i) \vee \Theta \end{aligned}$$

A consequence of the above observation is that we do not need to detect when we have reached a fixed point in order to output an invariant.

As a technical point about implementation of the above greatest fixed point computation in SAL, we mention that we break up the (possibly infinite) state space of the system into finitely many (disjoint) control states. Thereafter, rather than working with the global invariants ϕ_i, we work with local invariants that hold at particular control states. The iterative greatest fixed point computation can now be seen as a method of generating invariants based on *affirmation* and *propagation* [6].

Note that rather than computing the greatest fixed point, if we performed the least fixed point computation, we would get the strongest invariant for any given system. The problem with least fixed points is that their computation does not converge as easily as those of greatest fixed points. Unlike greatest fixed points, the intermediate predicates in the computation of the least fixed point are not invariants. We are currently investigating approaches based on widening to compute invariants in a convergent manner using least fixed points [8].

The techniques described so far are noncompositional since they examine all the transitions of the given system. We use a novel composition rule defined in [29] allowing local invariants of each of the modules to be composed into global invariants for the whole system. This composition rule allows us to generate stronger invariants than the invariants generated by the techniques described in [6,7]. The generated invariants allows us to obtain boolean abstractions of the analyzed system using the incremental analysis techniques presented in [29].

3.3 Slicing

Program analyses like slicing can help remove code irrelevant to the property under consideration from the input transition system which may result in a reduced state-space, thus easing the computational needs of subsequent formal analysis efforts. Our slicing tool [18] accepts an input transition system which may be synchronously or asynchronously composed of multiple modules written in SAL and the property under verification. The property under verification is converted into a slicing criterion and the input transition system is sliced with respect to this slicing criterion. The slicing criterion is merely a set of local/output variables of a subset of the modules in the input SAL program that are not relevant to the property. The output of the slicing algorithm is another SAL program similarly composed of modules wherein irrelevant code manipulating irrelevant variables from each module has been sliced out. For every input module there will be an output module, empty or otherwise. In a nutshell the slicing algorithm does a dependency analysis of each module and computes backward transitive closure of the dependencies. This transitive closure would take into consideration only a subset of all transitions in the module. We call these transitions observable and the remaining transitions are called τ or silent transitions. We replace silent transitions with skips.

We are currently investigating reduction techniques that are simpler than slicing and also ones that are more aggressive. One example is the cone-of-influence reduction where the slicing criterion is a set of variables V, and the reduction computes a transition system that includes all the variables in the transitive closure of V

given by the dependencies between variables [21]. In comparison with slicing, the cone-of-influence reduction is insensitive to control and is therefore easier to compute but generally not as efficient at pruning irrelevance. Slicing preserves program behavior with respect to the slicing criterion. One could obtain a more dramatic reduction by admitting slices that admitted more behaviors by introducing nondeterminism. Such aggressive slicing would be needed for example to abstract away from the internal behavior of a transition system within its critical section for the purpose of verifying mutual exclusion. Slicing for concurrent systems with respect to temporal properties has been investigated by Dwyer and Hatcliff [16].

3.4 Connecting InVeSt with SAL

So far we have described specialized SAL components that provide core features for the analysis of concurrent systems, but we have also integrated the standalone InVeSt [5] into the SAL framework. Besides compositional techniques for constructing abstraction and features for generating counterexamples from failed verification attempts, InVeSt introduces alternative methods for invariant generation to SAL. InVeSt not only serves as a backend tool for SAL but also has been connected to the IF laboratory [10], Aldebaran [9], TGV [17] and Kronos [15].

The salient feature of InVeSt is that it combines the algorithmic with the deductive approaches to program verification in two different ways. First, it integrates the principles underlying the algorithmic (e.g. [11,28]) and the deductive methods (e.g. [24]) in the sense that it uses fixed point calculation as in the algorithmic approach but also the reduction of the invariance problem to a set of first-order formulas as in the deductive approach. Second, it integrates the theorem prover PVS [27] with the model checker SMV [25] through the automatic computation of finite abstractions. That is, it provides the ability to automatically compute finite abstractions of infinite state systems which are then analyzed by SMV or, alternatively, by the model checker of PVS. Furthermore, InVeSt supports the proof of invariance properties using the method based on induction and auxiliary invariants (e.g. [24]) as well as a method based on abstraction techniques [2, 12–14, 21, 22]. InVeSt uses PVS as a backend tool and depends heavily on its theorem proving capabilities for deciding the myriad verification conditions.

3.4.1 Abstraction

InVeSt provides also a capability that computes an abstract system from a given concrete system and an abstraction function. The method underlying this technique is presented in [4]. The main features of this method is that it is automatic and compositional. It computes an abstract system $S^a = S^1_\alpha \parallel \cdots \parallel S^n_\alpha$, for a given system $S = S^1 \parallel \cdots \parallel S^n$ and abstraction function α, such that S simulates S_α is guaranteed by the construction. Hence, by known preservation results, if S_α satisfies an invariant φ then S satisfies the invariant $\alpha^{-1}(\varphi)$. Since the produced abstract system is not given by a graph but in a programming language, one still can apply all the known methods for avoiding the state explosion problem while analyzing S_α. Moreover, it generates an abstract system which has the same structure as the concrete one. This gives the ability to apply further abstractions and techniques to reduce the state explosion problem and facilitates the debugging of the concrete system. The computed abstract system is optionally represented in the specification language of PVS or in that of SMV.

The basic idea behind our method of computing abstractions is simple. In order to construct an abstraction of S, we construct for each concrete transition τ_c an abstract transition τ_a. To construct τ_a we proceed by elimination starting from the universal relation, which relates every abstract state to every abstract state, and eliminate pairs of abstract states in a conservative way, that is, it is guaranteed that after elimination of a pair the obtained transition is still an abstraction of τ_c. To check whether a pair (a, a') of abstract states can be eliminated we have to check that the concrete transition τ_c does not lead from any state c with $\alpha(c) = a$ to any state c' with $\alpha(c') = a'$. This amounts to proving a Hoare triple. The elimination method is in general too complex. Therefore, we combine it with three techniques that allow many fewer Hoare triples to be checked. These techniques are based on partitioning the set of abstract variables, using substitutions, and a new preservation result which allows to use the invariant to be proved during the construction process of the abstract system.

We implemented our method using the theorem prover PVS [27] to check the Hoare triples generated by the elimination method. The first-order formulas corresponding to these Hoare triples are constructed automatically and a strategy that is given by the user is applied. In [1] we developed also a general analysis methodology for *heterogeneous* infinite-state models, extended automata operating on variables which may range over several different domains, based on combining abstraction and symbolic reachability analysis.

3.4.2 Generation of Invariants

There are two different way to generate invariants in InVeSt. First, we use calculation of pre-fixed points by applying the body of the backward procedure a finite number of times and use techniques for the automatic generation of invariants (cf. [3]) to support the search for auxiliary invariants. The tool provides strategies which allow derivation of *local invariants*, that is, predicates attached to control locations and which are satisfied whenever the computation reaches the corresponding control point. InVeSt includes strategies for deriving local invariants for sequential systems as well as a composition principle that allows combination of invariants generated for sequential systems to obtain invariants of a composed system. Consider a composed system $S_1 \parallel S_2$ and control locations l_1 and l_2 of S_1 and S_2, respectively. Suppose that we generated the local invariants P_1 and P_2 at l_1 and l_2, respectively. Let us call P_i *interference independent*, if P_i does not contain a free variable that is written by S_j with $j \neq i$. Then, depending on whether P_i is interference independent we compose the local invariants P_1 and P_2 to obtain a local invariant at (l_1, l_2) as follows: if P_i is interference independent, then we can affirm that P_i is an invariant at (l_1, l_2) and if both P_1 and P_2 are interference dependent, then $P_1 \vee P_2$ is an invariant at (l_1, l_2). This composition principle proved to be useful in the examples we considered. However, examples showed that predicates obtained by this composition principle can become very large. Therefore, we also consider the alternative option where local invariants are not composed until they are needed in a verification condition. Thus, we assign to each component of the system two lists of local invariants. The first corresponds to interference independent local invariants and the second to interference dependent ones. Then, when a verification condition is considered, we use heuristics to determine which local invariants are useful when discharging the verification condition. A useful heuristic concerns the case when the verification condition is of the form $(pc(1) = l_1 \wedge pc(2) = l_2) \Rightarrow \phi$, where $pc(1) = l_1 \wedge pc(2) = l_2$ asserts that computation is at the local control locations l_1 and l_2. In this case, we combine the local invariants associated to l_1 and l_2 and add the result to the left hand side of the implication.

Second, we use abstraction generating invariants at the concrete level: Let $\mathcal{S}_{\alpha_1}$ the result of the abstraction of a concrete system $\mathcal{S}$, the set of reachable states denoted by $Reach(\mathcal{S}_{\alpha_1})$ is an invariant of $\mathcal{S}_{\alpha_1}$ (the strongest one including the initial configurations in fact). We developed a method that extract the formula which characterizes the reachable states from the BDD. Hence, $\alpha_1^{-1}(Reach(\mathcal{S}_{\alpha_1}))$ is an invariant of the concrete model $\mathcal{S}$. This invariant can be used to strengthen φ and show that it is an invariant of $\mathcal{S}$.

3.4.3 Analysis of Counterexamples

The generation of the abstract system is *completely automatic* and compositional as we consider transition by transition. Thus, for each concrete transition we obtain an abstract transition (which might be nondeterministic). This is a very important property of our method, since it enables the debugging of the concrete system or alternatively enhancing the abstraction function. Indeed, the constructed abstract system may not satisfy the desired property, for three possible reasons:

1. The concrete system does not satisfy the invariant,
2. The abstraction function is not suitable for proving the invariant, or
3. The proof strategies provided are too weak.

Now, a model checker such as SMV provides a trace as a counterexample, if the abstract system does not satisfy the abstract invariant. Since we have a clear correspondence between abstract and concrete transitions, we can examine the trace and find out which of the three reasons listed above is the case. In particular if the concrete system does not satisfy the invariant then we can transform the trace given by SMV to a concrete trace, thus generating a concrete counterexample.

3.5 Predicate/Boolean Abstraction

In addition to the InVeSt abstraction mechanisms, we implemented boolean abstraction of SAL specifications. We use the boolean abstraction scheme defined in [19] that uses predicates over concrete variables as abstract variables to abstract infinite or large state systems into finite state systems analyzable by model checking. The advantage of using boolean abstractions can be summarized as follows:

- Any abstraction to a finite state system can be expressed as a boolean abstraction.
- The abstract transition relation can be represented symbolically using Binary Decision Diagram (BDDs). Thus, efficient symbolic model checking [25] can be effectively applied.
- We have defined in [30] an efficient algorithm for the construction of boolean abstractions. We also designed an efficient refinement technique that allows us to refine automatically an already constructed abstraction until the property of interest is proved or a counter-example is generated.

- Abstraction followed by model checking and successive refinement is an efficient and more powerful alternative to invariant generation techniques such as the ones presented in [6,7].

3.5.1 Automatic Construction of Boolean Abstractions

The automatic abstraction module takes as input a SAL basemodule and a set of predicates defining the boolean abstraction. Using the algorithm in [30] we automatically construct the corresponding abstract transition system. This process relies heavily on the PVS decision procedures.

```
...
INPUT   x: integer
OUTPUT  y, z: integer

INITIALIZATION
  TRUE -->  INIT(x) = 0;
            INIT(y) = 0;
            INIT(z) = y;

TRANSITION
  NOT(x > 0)   -->  y' = y + 1
  []    z > 0  -->  z' = y - 1, y' = 0
...
```

Figure 3. Concrete Module.

Figure 3 and 4 display a simple SAL module and its abstraction where the boolean variables B1, B2 and B3 correspond to the predicates $x > 0$, $y > 0$, and $z > 0$. Notice that the assignment to B3 is nondeterministically chosen from the set {TRUE, FALSE}.

3.5.2 Explicit Model Checking

Finite-state SAL modules can be translated to SMV for model checking as explained above. However, model checkers usually do not allow to access their internal data structures where intermediate computation steps of the model-checking process can be exploited. For this reason, we implemented an efficient explicit-state model checker for SAL systems obtained by boolean abstraction. The abstract SAL description is translated into an executable Lisp code that performs the explicit state model checking procedure allowing us to explore about twenty thousand states a second. This procedure builds an abstract state graph that can be exploited for further analysis. Furthermore, additional abstractions can be applied on the fly while the abstract state graph is being built.

```
...
INPUT  B1: boolean
OUTPUT B2,B3: boolean

INITIALIZATION
  TRUE -->  INIT(B1) = FALSE;
            INIT(B2) = FALSE;
            INIT(B3) = FALSE;

TRANSITION
    NOT(B1) --> B2'=F
 [] B3 --> B2'=T, B3'= { TRUE, FALSE }
...
```

Figure 4. Abstract Module.

3.5.3 Automatic Refinement of Abstractions

When model checking fails to establish the property of interest, we use the results developed in [29,30] to decide whether the constructed abstraction is too coarse and needs to be refined, or that the property is violated in the concrete system and that the ge nerated counterexample corresponds indeed to an execution of the concrete system violating the property. This is done by examining the generated abstract state graph. The refinement technique computes the precondition to a transition where nondeterministic assignments occur. The preconditions corresponding to the cases where the variables get either TRUE or FALSE define two predicates that are used as new abstract variables. The following transition from the example

```
B3 --> B2'=TRUE, B3'= {TRUE, FALSE}
```

can be automatically refined to

```
B3 --> B2'=TRUE, B3'=B4 ,
       B4'=FALSE, B5' = FALSE
```

where B4 and B5 correspond to the predicates y=1 and y>1, respectively.

4 Conclusions

SAL is a tool that combines techniques from static analysis, model checking, and theorem proving in a truly integrated environment. Currently, its core is realized as an extension of the PVS system and has a well-defined interface for coupling specialized analysis tools. So

far, we have been focusing on developing and connecting back-end tools for validating SAL specifications by means of animation, theorem proving, and model checking, and also for computing abstractions, slices, and invariants of SAL modules. There are as yet no automated translators into the SAL language. Primary candidates are translators for source languages such as Java, Verilog, Esterel, Statecharts, or SDL. Since SAL is an open system with well-defined interfaces, however, we hope others will write those if the rest of the system proves effective.

We are currently completing the implementation of the SAL prototype which includes a parser, typechecker, a slicer, an invariant generator, the connection to InVeSt, and translators to SMV and PVS. We expect to release the prototype SAL system in mid-2000.

Although our experience with the combined power of several forms of mechanized formal analysis in the SAL system is still rather limited, we predict that proofs and refutations of concurrent systems that currently require significant human effort will soon become routine calculations.

References

[1] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [20], pages 146–159.

[2] S. Bensalem, A. Bouajjani, C. Loiseau, and J. Sifakis. Property preserving simulations. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification'92*, volume 663 of *LNCS*, pages 260–273. Springer-Verlag, 1992.

[3] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

[4] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems automatically and compositionally. In A. J. Hu and M. Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 319–331. Springer-Verlag, 1998.

[5] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In A. J. Hu and M. Y. vardi, editors, *Computer Aided Verification*, volume 1427 of *LNCS*, pages 505–510. Springer-Verlag, 1998.

[6] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[7] N. Bjørner, I. A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[8] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993. Vol. 735 of *Lecture Notes in Computer Science*, Springer-Verlag.

[9] M. Bozga, J. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the Aldebaran toolset. *Software Tools and Technology Transfer journal*, 1998.

[10] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *Proceedings of FM'99, Toulouse, France*, LNCS, 1999.

[11] E. Clarke, E. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symp. of Prog. Lang.* ACM Press, 1983.

[12] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.

[13] D. Dams. *Abstract interpretation and partition refinement for model checking*. PhD thesis, Technical University of Eindhoven, 1996.

[14] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 (PROCOMET)*. IFIP Transactions, North-Holland/Elsevier, 1994.

[15] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. FORTE'94*, Berne, Switzerland, Oct. 1994.

[16] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Jan. 1999.

[17] J.-C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (Rutgers University, New Brunswick, NJ, USA)*, volume 1102 of *LNCS*. Springer Verlag, 1996. Also available as INRIA Research Report RR-2987.

[18] V. Ganesh, H. Saïdi, and N. Shankar. Slicing SAL. Draft, 1999.

[19] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[20] N. Halbwachs and D. Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[21] R. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[22] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon, 1993.

[23] G. Lüttgen and V. Carreño. Analyzing mode confusion via model checking. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking (SPIN '99)*, volume 1680 of *Lecture Notes in Computer Science*, pages 120–135, Toulouse, France, September 1999. Springer-Verlag.

[24] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[25] K. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Boston, 1993.

[26] C. Muñoz and J. Rushby. Structural embeddings: Mechanization with method. In *Proceedings of the World Congress on Formal Methods FM 99*, volume 1708 of *LNCS*, pages 452–471, 1999.

[27] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[28] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Sym. on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[29] H. Saïdi. Modular and incremental analysis of concurrent software systems. In *14th IEEE International Conference on Automated Software Engineering*, Oct. 1999.

[30] H. Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [20], pages 443–454.

[31] The SAL Group. The SAL intermediate language. Available at: `http://sal.csl.sri.com/`, 1999.

Invited paper at ASM2000: Abstract State Machine Workshop, Monte Verita, Switzerland, March 2000.

Symbolic Analysis of Transition Systems*

Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

*This work was funded by the Defence Advanced Research Projects Agency under Contract NO. F30603-96-C-0204, and NSF Grants No. CCR-9712383 and CCR-9509931. The SAL project is a combined effort between SRI International, Stanford University, and the University of California, Berkeley.

Abstract

We give a brief overview of the *Symbolic Analysis Laboratory* (SAL) project. SAL is a verification framework that is directed at analyzing properties of transition systems by combining tools for program analysis, model checking, and theorem proving. SAL is built around a small intermediate language that serves as a semantic representation for transition systems that can be used to drive the various analysis tools.

1 Introduction

The transition system model of a program consists of a state type, an initialization predicate on this state type, and a binary next-state relation. The execution of a program starts in a state satisfying the initialization predicate so that each state and its successor state satisfy the next-state relation. Transition systems are a simple low-level model that have none of the semantic complications of high-level programming languages. Constructs such as branches, loops, and procedure calls can be modelled within a transition system through the use of explicit control variables. The transition system model forms the basis of several formalisms for several popular formalisms including UNITY [11], TLA [28], SPL [30], and ASMs [21]. It also underlies verification tools such as SMV [32], Murphi [18], and STeP [31].

If we focus our attention on the verification of properties of transition systems, we find that even this simple model poses some serious challenges. The verification of transition systems is performed by showing that the system satisfies an invariance or progress property, or that it refines another transition system. It is easy to write out proof rules for the verification of such properties but the actual application of these proof rules requires considerable human ingenuity. For example, the verification of invariance properties requires that the invariant be inductive, i.e., preserved by each transition. A valid invariant might need to be strengthened before it can be shown to be inductive. Fairness constraints and progress measures have to be employed for demonstrating progress properties. It takes a fair amount of effort and ingenuity to come up with suitable invariant strengthenings and progress measures.

Methods like model checking [12] that are based on state-space exploration have the advantage that they are largely automatic and seldom require the fine-grain interaction seen with deductive methods. Since these methods typically explore the reachable state space (i.e., the strongest invariant), there is no need for invariant strengthening. Progress measures are also irrelevant since the size of the whole state space is bounded. However, model checking methods apply only to a limited class of systems that possess small, essentially finite state spaces.

Theorem proving or model checking are not by themselves adequate for effective verification. It is necessary to combine the expressiveness of the deductive methods with the automation given by model checking. This way, small, finite-state systems can be directly verified using model checking. For larger, possibly infinite-state systems, theorem proving can be used to construct *property-preserving abstractions* over a smaller state space. Such abstractions convert data-specific characteristics of a computation into control-specific ones. The finite-state model constructed by means of abstraction can be analyzed using model checking. It is easy to actually compute the properties of a system from a finite-state approximation and map these properties back to the original system.

We give an overview of an ongoing effort aimed at constructing a general framework for the integration of theorem proving, model checking, and program analysis. We use the term *symbolic analysis* to refer to the integration of these

analysis techniques since they all employ representations based on symbolic logic to carry out a symbolic interpretation of program behavior. The framework also emphasizes *analysis*, i.e., the extraction of a large number of useful properties, over *correctness* which is the demonstration of a small number of important properties. The framework is called the *Symbolic Analysis Laboratory* (SAL). We motivate the need for symbolic analysis and describe the architecture and intermediate language of SAL.

2 A Motivating Example

We use a very simple and artificial example to illustrate how symbolic analysis can bring about a synergistic combination of theorem proving, model checking, and program analysis. The example consists of a transition system with a state contain a (control) variable PC ranging over the scalar type $\{inc, dec\}$, and two integer variables B and C. Initially, control is in state inc and the variables B and C are set to zero. There are three transition rules shown below as guarded commands:

1. When $PC = inc$, then B is incremented by two, C is set to zero, and control is transferred to state dec.

$$PC = inc \longrightarrow B' = B + 2; C' = 0; PC' = dec;$$

2. When $PC = dec$, B is decremented by two, and C is incremented by one, and control is transferred to state $decl$.

$$PC = dec \wedge B > 0 \longrightarrow B' = B - 2; C' = C + 1; PC' = inc;$$

3. Same as transition rule 2, but control stays in dec.

$$PC = dec \wedge B > 0 \longrightarrow B' = B - 2; C' = C + 1;$$

There is also an implicit stuttering transition from state dec to itself when none of the guards of the other transitions holds, i.e., when $B \leq 0$. Since the inc state has a transition with a guard that is always true, there is no need for a stuttering transition on inc. The transition system is shown diagrammatically in Figure 1.

The transition system *Twos* satisfies a number of interesting invariants.

1. B is always an even number.
2. B and C are always non-negative.
3. B is always either 0 or 2.
4. B is always 0 in state inc.
5. C is always either 0 or 1.

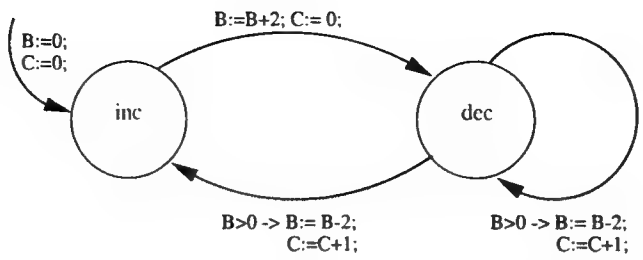


Figure 1: A Simple Transition System: Twos

6. In state *dec*, $B = 2$ iff $C = 0$.

The purpose of symbolic analysis is to find and validate such properties with a high degree of automation and minimal human guidance and intervention. While efficient automation is essential for analyzing large transition systems, the intended outcome of symbolic analysis is human insight. The analysis should therefore not rule out human interaction.

3 Some Symbolic Analysis Techniques

We enumerate some symbolic analysis techniques and assess their utility on the *Twos* example. For this purpose, we focus on the invariant (1) below.

$$B = 0 \vee B = 2 \tag{1}$$

Note that the transition system *Twos* is a potentially infinite state system since variables B and C range over the integers.

Some mathematical preliminaries are in order. A transition system P is given by a pair $\langle I_P, N_P \rangle$ consisting of an initialization predicate on states I_P, and a binary next-state relation on states N_P. We constrain the next-state relation N to be total so that $\forall s : \exists s' : N(s, s')$. The metavariables s, s' range over states. We treat a set of states as equivalent to its characteristic predicate. The boolean connectives $\wedge$, $\vee$, $\supset$, are lifted from the booleans to the level of predicates and correspond to the set-theoretic operations $\cap$, $\cup$, and $\subseteq$, respectively. An assertion is a predicate on states. The metavariables ϕ, ψ range over assertions. A predicate transformer is a map from predicates to predicates. A monotone predicate transformer τ preserves the subset or implication ordering on predicates so that if $\phi \supset \psi$, then $\tau(\phi) \supset \tau(\psi)$. The fixed point of a monotone predicate transformer τ is an assertion ϕ such that $\phi \equiv \tau(\phi)$. As a consequence of the Tarski–Knaster theorem, every monotone predicate transformer has a least fixed point $lfp(\tau)$ and a greatest fixed point $gfp(\tau)$ such that

$$lfp(\tau) = \tau(lfp(\tau)) \supset gfp(\tau) = \tau(gfp(\tau)).$$

Let $\bot$ represent the empty set of states, $\top$ the set of all states, and ω the set of natural numbers. If the state space is finite, then the least fixed point $lfp(\tau)$ can be calculated as

$$\bot \vee \tau(\bot) \vee \tau^2(\bot) \vee \ldots \vee \tau^n(\bot)$$

for some n, and similarly, $gfp(\tau)$ can be calculated as

$$\top \wedge \tau(\top) \wedge \tau^2(\top) \wedge \ldots \wedge \tau^n(\top),$$

for some n.

If τ is $\vee$-continuous (i.e., $\tau(\bigvee_{i\in\omega} \phi_i) = \bigvee_{i\in\omega} \tau(\phi_i)$ for ϕ_i such that whenever $i < j$, $\phi_i \supset \phi_j$), then

$$lfp(\tau) = \bigvee_{i\in\omega} \tau^i(\bot) \tag{2}$$

Similarly, if τ is $\wedge$-continuous (i.e., $\tau(\bigwedge_{i\in\omega} \phi_i) = \bigwedge_{i\in\omega} \tau(\phi_i)$ for ϕ_i such that whenever $i < j$, $\phi_j \supset \phi_i$), then

$$gfp(\tau) = \bigwedge_{i\in\omega} \tau^i(\top) \tag{3}$$

Equations (2) and (3) provide an iterative way of computing the least and greatest fixed points but these on infinite-state spaces, the computations might not converge in a bounded number of steps.

Typical examples of monotone predicate transformers include

1. Strongest postcondition of a transition relation N, $sp(N)$, which is defined as
$$sp(N)(\phi) \equiv (\exists s' : \phi(s') \wedge N(s', s)).$$
2. Strongest postcondition of a transition system P, $sp(P)$ is defined as
$$sp(P)(\phi) \equiv I_P \vee sp(N_P)(\phi).$$
3. Weakest precondition of a transition relation N, $wp(N)$ is defined as
$$wp(N)(\phi) \equiv (\forall s' : N(s, s') \supset \phi(s)).$$

3.1 Invariant Proving

The invariance rule is the most heavily used proof rule in any program logic [24, 33]. Given a transition system P as a pair $\langle I_P, N_P \rangle$, consisting of an initialization I_P and a next-state relation N_P, the invariance rule usually has the form:

$$\frac{\begin{array}{c}\vdash \psi(s) \supset \phi(s) \\ \vdash I_P(s) \supset \psi(s) \\ \vdash \psi(s_0) \wedge N_P(s_0, s_1) \supset \psi(s_1)\end{array}}{P \models \textbf{invariant } \phi}$$

In this rule, the assertion ψ is a strengthening of the assertion ϕ. Such a strengthening is needed since the assertion ϕ may not be inductive, i.e., satisfy the premises $\vdash I_P(s) \supset \phi(s)$ and $\vdash \phi(s_0) \wedge N_P(s_0, s_1) \supset \phi(s_1)$.

In the *Twos* example, the invariant (1) is not inductive. It fails because it is not preserved by transition 1 since we cannot establish

$$\begin{array}{ll} \vdash & (PC = inc \wedge (B = 0 \vee B = 2)) \\ \wedge & (PC = inc \wedge B' = B + 2 \wedge C' = 0 \wedge PC' = dec) \\ \supset & (B' = 0 \vee B' = 2). \end{array}$$

The invariant has to be strengthened with the observation that when $PC = inc$, B is always 0 so that it now reads

$$B = 0 \vee (PC \neq inc \wedge B = 2). \tag{4}$$

The strengthened invariant (4) is inductive. The need for invariant strengthening in program proofs is the key disadvantage of the deductive methods with respect to model checking. Quite a lot of effort is needed to turn a putative invariant into an inductive one. Once an invariant has been strengthened in this manner, it can contain a large number of conjuncts that generate a case explosion in the proof. Much of the focus of symbolic analysis is on supplementing deductive verification with the means of automatically obtaining useful invariants and invariant strengthenings.

3.2 Enumerative Model Checking

The early approaches to model checking were based on the feasibility of computing fixed point properties for finite-state systems. The reachable states of a finite-states can be computed by starting from the set of initial states and exploring the states reachable in n consecutive transitions. Any property that holds on all the reachable states is a valid invariant. There are many variations on this basic theme. Many modern enumerative model checkers such as Murphi [18] and SPIN [25] carry out a depth-first search exploration of the transition graph while maintaing a hash-table to record states that have already been visited. In SPIN, the LTL model checking problem is transformed into one of emptiness for ω-automata, i.e., automata that recognize infinite strings [38, 20].

In enumerative model checking, properties written in a branching-time temporal logic CTL can be verified in time proportional to $N \times F$ where N is the size of the transition graph and F the size of the temporal formula. Model checking linear-time temporal logic formulas is more expensive and takes time proportional to $N \times 2^F$ where N is the size of the model and F is of the formula.

The *Twos* example succumbs rather fortuitously to enumerative model checking. Even though the potential state space of *Twos* is unbounded, only a bounded part of the state space is reachable since B is either 0 or 2, and C is either 0 or 1. The success of enumerative model checking is somewhat anomalous since this method is unlikely to terminate on typical infinite-state systems. Even on finite-state systems, an enumerative check is unlikely to succeed because the

size of the searchable state space can be exponential in the size of the program state. Still, enumerative model checking is an effective debugging technique that can often detect and display simple counterexamples when a property fails.

3.3 Symbolic Model Checking

The use of symbolic representation for the state sets was proposed in order to combat the state explosion problem in enumerative model checking [10, 32]. A symbolic representation for boolean functions based on binary decision diagrams (BDDs) [9] has proved particularly successful. A finite state can be represented as a bit-vector. Then sets of bit-vectors are just boolean functions and can be represented as BDDs. In particular, the initial set, a given invariant claim, the transition relation, and the reachable state set, can all be represented as BDDs. The BDD operations can be used to compute images of state sets with respect to the transition relation. This allows predicate transformers such as strongest postcondition and weakest precondition to be applied to the BDD representation of a state set. The reachable state set can be computed by means of a fixed point iteration of the strongest postcondition computation starting from the initial state set. Every intermediate iteration of the reachable state set is also represented as a BDD. There are several advantages to the use of BDDs. Sometimes even sets of large cardinality might have compact symbolic representations. BDDs are a canonical representation for boolean functions so that equivalence tests are cheap. BDDs are especially good at handling the boolean quantification that is needed in the image computations. Automata-theoretic methods can also be represented in symbolic form. Some symbolic model checkers include SMV [32]

Such symbolic representations do require the state to be explicitly finite. This means that the *Twos* example cannot be coded directly in a form that can be directly understood by a symbolic model checker. Some work has to be done in order to reduce the problem to finite-state form so that it can be handled by a symbolic model checker.

3.4 Invariant Generation

Automatic invariant generation techniques have been studied since the 1970s [16, 19, 27, 37], and more recently in the work of Bjørner, Browne, and Manna [8], and Bensalem, Lakhnech, and Saïdi [6, 34, 4].

As in model checking, the basic operation in invariant generation is that of taking the strongest postcondition or weakest precondition of a state set X with respect to the transition relation N. Some of the techniques for computing invariants are described briefly below.

Least Fixed Point of the Strongest Postcondition. The invariant computed here corresponds to the reachability state set. It is computed by starting with an initial symbolic representation of the initial state set given by the program. This set is successively enlarged by taking its image under

the strongest postcondition operation until a fixed point is reached, i.e., no new elements are added to the set. We term this method LFP-SP. It yields a symbolic representation of the set of reachable states which is the strongest invariant. However, LFP-SP computation often does not terminate since the computation might not converge to a fixed point in a finite number of steps. Take, for example, a program that successively increments by one, a variable x that is initially zero. This program has a least fixed point, i.e., x is in the set of natural numbers, but the iterative computation does not converge.

For the *Twos* example, the LFP-SP computation does terminate with the desired invariant as seen in the calculation below.

$$\begin{aligned} Inv^0 &= (PC = inc \wedge B = 0 \wedge C = 0) \\ Inv^1 &= Inv^0 \vee (PC = dec \wedge B = 2 \wedge C = 0) \\ Inv^2 &= Inv^1 \vee (B = 0 \wedge C = 1) \\ Inv^3 &= (B = 0 \wedge C = 1) \vee (PC = dec \wedge B = 2 \wedge C = 0) \\ &= Inv^2 \end{aligned}$$

The resulting invariant easily implies the strengthened inductive invariant (4). The LFP-SP computation terminates precisely because the reachable state set is bounded. In more typical examples, approximation techniques based on *widening* will be needed to accelerate the convergence of the least fixed point computation.

Greatest Fixed Point of the Strongest Postcondition. The greatest fixed point iteration starts with the entire state space and strengthens it in each iteration by excluding states that are definitely unreachable. This approach, which we call GFP-SP, yields a weaker invariant than the least fixed point computation. The GFP-SP computation also need not terminate. Even when it does terminate, the resulting invariant might not be strong enough. In the case of the program with single integer variable x that is initially zero and incremented by one in each transition, the GFP-SP computation returns the trivial invariant **true**. However the GFP-SP method has the advantage that it can be made to converge more easily than the LFP-SP method, and any intermediate step in the computation already yields a valid invariant.

The greatest fixed point invariant computation for *Twos* (ignoring the variable C) can be carried out as follows. Here $Inv^i(pc)$ represents the i iteration of the invariant for control state pc.

$$\begin{aligned} Inv^0(inc) &= (B = 0 \vee B \geq -1) = (B \geq -1) \\ Inv^0(sub) &= \textbf{true} \end{aligned}$$

$$\begin{aligned} Inv^1(inc) &= (B \geq -1) \\ Inv^1(sub) &= (B \geq 1 \vee B \geq -1) = (B \geq -1) \end{aligned}$$

$$Inv^2(inc) = (B \geq -1)$$
$$Inv^2(sub) = (B \geq -1)$$

The invariant $B \geq -1$ is not all that useful since this information contributes nothing to the invariants that we wish to establish. Still, the GFP-SP method is not without value. It is especially useful for propagating known invariants. For example, if we start the iteration with invariant (1), then we can use the GFP-SP method to deduce that the strengthened invariant (4).

Greatest Fixed Point of the Weakest Precondition. Both LFP-SP and GFP-SP compute inductive invariants that are valid, whereas the method GFP-WP takes a putative invariant and strengthens it in order to make it inductive. The computation starts with a putative invariant S, and successively applies the weakest precondition operation wp(P)(S) to it. If this computation terminates, then either the resulting assertion is a strengthening of the original invariant that is also inductive, or the given invariant is shown to be invalid.

With the *Twos* example, the weakest precondition with respect to the putative invariant (1) yields the strengthened invariant (4).

3.5 Abstract Interpretation.

Many of the invariant generation techniques are already examples of abstract interpretation which is a general framework for lifting program execution from the concrete domain of values to a more abstract domain of properties. Examples of abstract interpretation include sign analysis (positive, negative, or zero) of variables, interval analysis (computing bounds on the range of values a variable can take), live variable analysis (the value of a variable at a control point might be used in the computation to follow), among many others.

We can apply an interval analysis to the *Twos* example. Initially, the interval for B is $[0, 0]$ for $PC = inc$. This yields an interval of $[2, 2]$ for B when $PC = dec$. In the next step, we have an approximation of $[0, 2]$ for B when $PC = dec$, and $[0, 0]$ when $PC = inc$. The next round, we get an approximation of $[-1, 0]$ for the range of B when $PC = inc$, and $[0, 2]$ for the range of B when $PC = dec$. At this point the computation converges, but the results of the analysis are still too approximate and do not discharge the invariant (1).

3.6 Property Preserving Abstractions

Since model checking is unable to cope with systems with infinite or large state spaces, abstraction has been studied as a technique for reducing the state space [14, 29, 35]. In data abstraction, a variable over an infinite or large type is reduced to one over a smaller type. The smaller type is essentially a quotient with respect to some equivalence relation of the larger type. For example, a variable ranging over the integers can be reduced to boolean form by considering only the parity (odd or even) of the numbers. *Predicate abstraction* is an extension of data abstraction that introduces boolean variables for predicates

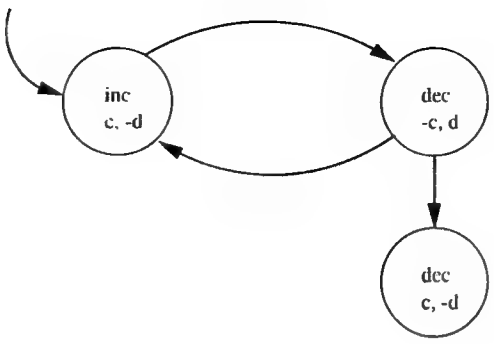


Figure 2: Abstract *Twos*

over a set of variables. For example, if x and y are two integer variables in a program, it is possible to abstract the program with respect to the predicates such as $x < y$, $x = y$. These variables are then replaced by boolean variables p and q such that p corresponds to the $x < y$ and q corresponds to $x = y$. Even though predicate abstraction introduces only boolean variables, it is possible to simulate a data abstraction of a variable to one of finite type by using a binary encoding of the finite type.

In general, an abstraction is given by means of a concretization map γ such that $\gamma(a)$ for an abstract variable a returns its concrete counterpart. In the case of the abstraction where $x < y$ is replaced by p and $x = y$ by q, $\gamma(a) = (x < y)$ and $\gamma(b) = (x = y)$. The more difficult direction is computing an abstraction $\alpha(C)$ given a concrete predicate C. The construction of α requires the use of theorem proving as described below.

There are also two ways of using abstractions in symbolic analysis. In one approach, the abstract reachability set [35, 17] is constructed by the following iteration

$$ARG(P)(s) = lfp(\alpha(I_P) \vee \alpha \circ sp(P) \circ \gamma).$$

We can then check if p is an invariant of P by verifying $\gamma(ARG(P)) \supset p$.

A second way of using abstraction is by actually constructing the abstracted version of the program and the property of interest [5, 15, 36]. This can be more efficient since the program and property are usually smaller than the abstract reachability graph.

In the *Twos* example, the predicate abstraction is suggested by the predicates $B = 0$ and $B = 2$ in the putative invariant. The abstract transition system by replacing the predicate $B = 0$ by c and $B = 2$ by d is shown in Figure 2.

The abstract transition system computed using predicate abstraction can easily be model checked to confirm that invariant (1) holds. The stronger invariant (4) can also be extracted from the reachable state space of the abstract transition system.

Predicate abstraction affords an effective integration of theorem proving and model checking where the former is used to construct a finite-state property-preserving abstraction that can be analyzed using the latter. The abstraction loses information so that a property can fail to hold in the abstract system even when its concrete counterpart is valid for the concrete system. In this case, the abstraction has to be refined by introducing further predicates for abstraction [5, 13].

4 SAL: A Symbolic Analysis Laboratory

We have already seen a catalog of symbolic analysis techniques. The idea of a symbolic analysis laboratory is to allow these techniques to coexist so that the analysis of a transition system can be carried out by successive applications of a combination of these techniques [3]. With such a combination of analysis techniques, one could envisage a verification methodology where

1. A cone-of-influence reduction is used to discard irrelevant variables.
2. Invariant generation is used to obtain small but useful invariants.
3. These invariants are used to obtain a reasonably accurate abstraction to a finite-state transition system.
4. Model checking is used to compute useful invariants of the finite-state abstraction.
5. The invariants computed by model checking over the abstraction are used propagated using invariant generation techniques.
6. This cycle can be repeated until no further useful information is forthcoming.

SAL provides a blackboard architecture for symbolic analysis where a collection of tools interact through a common intermediate language for transition systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this intermediate language and the analysis results are fed back to this intermediate level. In order to analyze systems that are written in a conventional source language, the transition system model of the source program has to be extracted and cast in the SAL intermediate language.[1] The model extracted in the SAL intermediate language essentially captures the transition system semantics of the original source program.

The SAL architecture is shown in Figure 3 The SAL architecture is constrained so that the different analysis tools do not communicate directly with each other, but do so through the SAL intermediate language. The interaction between the tools must therefore be at a coarse level of granularity, namely in

[1] We are currently working on a translator from a subset of Verilog to SAL, and another from a subset of Java to SAL.

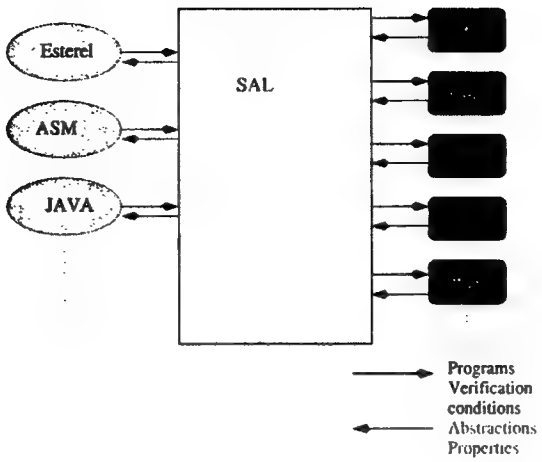


Figure 3: The Architecture of SAL

terms of transition systems, their properties, and property-preserving transformations between transition systems. Allowing the tools to communicate directly to each other would require a quadratic number of different maps (for a given number of tools) between these analysis tools.

4.1 The SAL Intermediate Language

The intermediate language for SAL[2] serves as

1. The target of translations from source languages.
2. The source for translations to the input formats of different analysis tools.
3. A medium for communication between different analysis tools.

The SAL intermediate language is based on languages and models such as SMV [32], Murphi [18], Reactive Modules [1], ASM [21], UNITY [11], and TLA [28], among others. The unit of specification in SAL is a context which contains declarations of types, constants, transition system modules, and assertions. A SAL module is a transition system unit. A basic SAL module is a state transition system where the state consists of *input*, *output*, *local*, and *global* variables, where

- An input variable to a module can be read but not written by the module.
- An output variable to a module can be read and written by the module, and only read by an external module.

[2]The SAL intermediate language was designed in collaboration with Prof. David Dill of Stanford, Prof. Tom Henzinger at UC Berkeley, and several colleagues at SRI, Stanford, and UC Berkeley.

- A local variable to a module can be read and written by the module, but is not read or written by the module.
- A global variable to a module can be read and written by the module as well as by an external module

A basic module also specifies the initialization and transition steps. These can be given by a combination of definitions or guarded commands. A definition is of the form $x = \textit{expression}$ or $x' = \textit{expression}$, where x' refers to the new value of variable x in a transition. A definition can also be given as a selection of the form $x' \in \textit{set}$ which means that the new value of x is nondeterministically selected from the value of of *set*. A guarded command is of the form $g \longrightarrow S$, where g is a boolean guard and S is a list of definitions of the form $x' = \textit{expression}$ or $x' \in \textit{set}$.

As in synchronous language such as Esterel [7] and Lustre [22], SAL allows *synchronous*, i.e., Mealy machine, interaction so that the new value of a local or output variable can be determined by the new value of a variable. Such interaction introduces the possibility of a causal cycle where each variable is defined to react synchronously to the other. Such causal cycles are ruled out by using static analysis to generate proof obligations demonstrating that such cycles are not reachable. The UNITY and ASM models do not admit such synchronous interaction since the new values of a variable in a transition are completely determined by the old values of the variables. SMV allows such interaction but the semantics is not clearly specified, particularly when causal cycles are possible. The Reactive Modules [1] language uses a static partial ordering on the variables that breaks causal loops by allowing synchronous interaction in one direction of the ordering but not the other. In TLA [28], two modules are composed by conjoining their transition relations. TLA allows synchronous interaction where causal loops can be resolved in any manner that is compatible with the conjunction of the transition relations is satisfied.

SAL modules can be composed

- *Synchronously*, so that $M_1 \| M_2$ is a module that takes M_1 and M_2 transitions in lockstep, or
- Asynchronously, so that $M_1 \, [] \, M_2$ is a module that takes an interleaving of M_1 and M_2 transitions.

There are rules that govern the usage of variables within a composition. Two modules engaged in a composition must not share output variables and nor should the output variables of one module overlap with the global variables of another. The modules can can share input and global variables, and the input variables of one module can be the output or global variables of the other. Two modules that share a global variable cannot be composed *synchronously*, since this might create a conflict when both modules attempt to write the variable synchronously. The rules governing composition allow systems to be analyzed modularly so that system properties can be composed from module properties [1].

The N-fold synchronous and asynchronous compositions of modules are also expressible in SAL. Module operations include those for hiding and renaming of variables. Any module defined by means of composition and other module operations can always be written as a single basic module, but with a significant loss of succinctness.

SAL does not contain features other than the rudimentary ones described above. There are no constructs for synchronization, synchronous message passing, or dynamic process creation. These have to explicitly implemented by means of the transition system mechanisms available in SAL. While these features are useful, their introduction into the language would place a greater burden on the analysis tools.

The SAL language is thus similar in spirit to Abstract State Machines [21] in that both serve as basic conceptual models for transition systems. However, machines described in SAL are not abstract compared with those in ASM notation since SAL is intended as a front-end to various popular model checking and program analysis tools.

5 Conclusions

Powerful automated verification technologies have become available in the form of model checkers for finite, timed, and hybrid systems, decision procedures, theorem provers, and static analyzers. Individually, these technologies are quite limited in the range of systems or properties they can handle with a high degree of automation. These technologies are complementary in the sense that one is powerful where the other is weak. Static analysis can derive properties by means of a syntactic analysis. Model checking is best suited for control-intensive systems. Theorem proving is most appropriate for verifying mathematical properties of the data domain. Symbolic analysis is aimed at achieving a synergistic integration of these analysis techniques. The unifying ideas are

1. The use of transition systems as a unifying model, and
2. Fixed point computations over symbolic representations as the unifying analysis scheme.
3. Abstraction as the key technique for reducing infinite-state systems to finite-state form.

Implementation work on the SAL framework is currently ongoing. The preliminary version of SAL consists of a parser, typechecker, causality checker, an invariant generator, translators from SAL to SMV and PVS, and some other tools. SAL is intended as an experimental framework for studying the ways in which different symbolic analysis techniques can be combined to achieve greater automation in the verification of transition systems.

Acknowledgments. Many collaborators and colleagues have contributed ideas and code to the SAL language and framework, including Saddek Bensalem, David Dill, Tom Henzinger, Luca de Alfaro, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Eli Singerman, Mandayam Srivas, Jens Skakkebæk, and Ashish Tiwari.

References

[1] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[2] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996.

[3] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, NASA Langley Research Center, Hampton, VA, June 2000. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[4] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.

[5] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [26], pages 319–331.

[6] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [2], pages 323–335.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[9] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[11] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[12] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[13] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification.* 2000. To appear.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[15] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [26], pages 293–304.

[16] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages.* January 1978.

[17] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [23], pages 160–171.

[18] David L. Dill. The Murϕ verification system. In Alur and Henzinger [2], pages 390–393.

[19] S. M. German and B. Wegbreit. A synthesizer for inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[20] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995.

[21] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, International Schools for Computer Scientists, pages 9–36. Oxford University Press, Oxford, UK, 1995.

[22] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[23] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[24] C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[25] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[26] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998.

[27] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[28] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[29] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[30] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[31] Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [2], pages 415–418.

[32] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[33] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, ACM, Providence, RI, November 1977.

[34] Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, Volume 1055 of Springer-Verlag *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996.

[35] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[36] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [23], pages 443–454.

[37] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.

[38] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science*. pages 332–344, IEEE Computer Society Press, 1986.

Invited Paper at CONCUR 2000: 11th International Conference on Concurrency Theory, Penn State University, State College, Pennsylvania, August 2000.

Combining Theorem Proving and Model Checking through Symbolic Analysis *

Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

*This work was funded by DARPA Contract No. F30602-96-C-0204 Order No. D855 and NSF Grants No. CCR-9712383 and CCR-9509931.

Abstract

Automated verification of concurrent systems is hindered by the fact that the state spaces are either infinite or too large for model checking, and the case analysis usually defeats theorem proving. Combinations of the two techniques have been tried with varying degrees of success. We argue for a specific combination where theorem proving is used to reduce verification problems to finite-state form, and model checking is used to explore properties of these reductions. This decomposition of the verification task forms the basis of the Symbolic Analysis Laboratory (SAL), a framework for combining different analysis tools for transition systems via a common intermediate language. We demonstrate how symbolic analysis can be an effective methodology for combining deduction and exploration.[1]

[1]The SAL project is a collaborative effort between Stanford University, SRI International, and the University of California, Berkeley.

1 Introduction

The verification of large-scale concurrent systems poses a difficult challenge in spite of the substantial recent progress in computer-aided verification. Technologies based on model checking [11] can typically handle systems with states that are no larger than about a hundred bits. Techniques such as symmetry and partial-order reductions, partitioned transition relations, infinite-state model checking, represent important advances toward ameliorating state explosion, but they have not dramatically increased the overall effectiveness of automated verification. Model checking does have one advantage: it needs only a modest amount of human guidance in terms of the problem description, possible variable orderings, and manually guided abstractions. Verification based on theorem proving, on the other hand, requires careful human control by way of suitable intermediate assertions, invariants, lemmas, and proofs. Can automated verification ever combine the automation of model checking with the generality of theorem proving?

It has often been argued that model checking and theorem proving could be combined so that the former is applied to control-intensive properties while the latter is invoked on data-intensive properties. Achieving an integration of theorem proving and model checking is not hard. Both techniques verify claims that look similar and it is possible to view model checking as a decision procedure for a well-defined fragment of a specification logic [41]. However, most systems contain a rich interaction between control and data so that there is no simple decomposition between data-intensive and control-intensive properties.

For the purpose of this paper, we view model checking as a technique for the verification of temporal properties of a program based on the exhaustive exploration of a transition graph represented in explicit or symbolic form. Model checking methods typically use graph algorithms, automata-theoretic constructions, or finite fixed point computations. Theorem proving is usually based on formalisms such as first-order or higher-order logic, and employs proof techniques such as induction, rewriting, simplification, and the use of decision procedures. Some infinite-state verifiers and semi-decision procedures can be classified as both deductive and model checking techniques, but this ambiguity can be overlooked for the present discussion.

We make several points regarding the use of theorem proving and model checking in the automated verification of concurrent systems:

1. *Correctness is over-rated.* The objective of verification is analysis, i.e., the accretion of useful observations regarding a system. Verifying correctness is an important form of analysis, but correctness is usually a big property of a system that is demonstrated by building on lots of small observations. If these small observations could be cheaply obtained, then the demonstration of larger properties would also be greatly simplified. The main drawback of correctness is its exactitude. The verification of a correctness claim can only either fail or succeed. There is no room for approximate answers or partial information.

2. *Theorem proving is under-rated.* Deduction remains the most appropriate technology for obtaining insightful, general, and reusable automation in the analysis of systems, particularly those that are too complex to be analyzed by a blunt instrument like model checking. Theorem proving can exploit the mathematical properties of the control and data structures underlying an algorithm in their fullest generality and abstractness

3. *Theorem proving and model checking are very similar techniques.* In the verification of transition systems, both techniques employ some representation for program assertions, they compute the image of the transition relation with respect to these assertions, and usually try to compute the least, greatest, or some intermediate fixed point assertion for the transition relation. The difference is that in theorem proving,

   - The image constructions are usually more complicated since they involve quantification in domains where quantifier elimination is either costly or impossible.
   - The least and greatest fixed points can seldom be effectively computed and human guidance is needed to suggest an intermediate fixed point.
   - Showing that one assertion is the consequence of another is typically undecidable and requires the use of lemmas and human insight.

4. *Theorem proving and model checking can be usefully integrated.* Such an integration requires a methodology that decomposes the verification task so that

   - *Deduction is used to construct valid finite-state abstractions of a system.* The construction of a property-preserving abstraction generates simple proof obligations that can be discharged, often fully automatically, using a theorem prover. These are typically assertions of the form: if property p holds in a state s from which there is a transition R to a state s', then property q holds in s'. Similar proof obligations arise during verification (in the form of verification conditions) but these are usually not valid and the assertions have to be strengthened in order to obtain provable verification conditions. While theorem proving is useful for examining the local consequence of properties, it is not very effective at deducing global consequences over a large program or around an iterative loop. Such computations can be extremely inefficient and the computation of fixed points around a loop rarely terminates.
   - *Exploration by means of model checking is used to calculate global properties of such abstractions.* This means that model checking is not used merely to validate or refute putative properties but is actually used to calculate interesting invariants that can be extracted from the reachability predicate or its approximations. Finite-state

exploration of large structures can also be inefficient but it is much easier to make finite-state computations converge efficiently.

- *Deduction is used to propagate the consequences of such properties.* For example, model checking on a finite-state abstraction might reveal an assertion $x > 5$ to hold at a program point simply because it was true initially and none of the intermediate transitions affected the value of x. If the program point has a successor state that can only be reached by a transition that increments x by 2, then we know that this successor state must satisfy the assertion $x > 7$. Such a consequence is easily deduced by theorem proving.

In summary, we advocate a verification methodology where deduction is employed in the local reasoning steps such as validating abstractions and propagating known properties, whereas model checking is used for deriving global consequences. In contrast, early attempts to integrate theorem proving and model checking were directed at using model checking as a decision procedure within a theorem prover. These attempts were not all that successful because it is not common to find finite-state subgoals within an infinite-state deductive verification.

2 Background

We review some of the background and previous work in the combined use of theorem proving and model checking techniques.

2.1 Model Checking as a Decision Procedure

Joyce and Seger combined the theorem prover HOL [22] with the symbolic trajectory evaluation tool Voss [28] by treating the circuits verified by Voss as uninterpreted constants in HOL. This integration is somewhat *ad hoc* since the definitions of the circuits verified by Voss are not available to HOL. Dingel and Filkorn [20] use a model checker to establish assume–guarantee properties of components and a theorem prover to discharge the proof obligations that arise when two components are composed. Rajan, Shankar, and Srivas [41] integrate a mu-calculus [40, 9] model checker [27] as a decision procedure for a fragment of the PVS higher-order logic corresponding to a finite mu-calculus. While this integration smoothly incorporates CTL and LTL model checking into PVS, the work needed to reduce a problem into model-checkable form can be substantial. This integration has recently been extended with an algorithm for constructing finite-state abstractions of mu-calculus expressions [45].[2]

[2]These features are part of PVS 2.3 which is accessible at the URL `pvs.csl.sri.com`.

2.2 Extending Model Checking with Lightweight Theorem Proving

Several alternative approaches to the integration of model checking and theorem proving have emerged in recent years. Some of these have taken the approach of supplementing a model checker with a proof assistant that provides rules for decomposing a verification goal into model-checkable subgoals. McMillan [36] in his work with Cadence SMV has extended the SMV model checker with the following decomposition rules that are used to reduce infinite-state systems to model-checkable finite-state ones.

1. *Temporal splitting*: Transforms a goal of the form $\Box(\forall i : A)$ into $\Box v = i \supset A$ for each i.

2. *Symmetry reduction*: Typically, the system being verified and the property are symmetric in the choice of i so that proving $\Box v = i \supset A$ for a single specific value for i is equivalent to proving it for each i. Examples of such symmetric choices include the memory address or the processor in the verification of multiprocessor cache consistency.

3. *Data abstraction*: Large or infinite datatypes can be reduced to small finite datatypes by suitably reinterpreting the operations on these datatypes. For example, with respect to the choice of i in temporal splitting, the remaining values of the datatype can be abstracted by a single value *non-i*.

4. *Compositional verification*: The verification of $P \| Q \models A \wedge B$ is decomposed as $P \models \neg(B \mathbf{U} \neg A)$ (B fails before A does) and $Q \models \neg(A \mathbf{U} \neg B)$. This allows different components to be separately verified up to time $t+1$ by assuming the other components to be correct up to time t.

These and other proof techniques have been used to verify an out-of-order processor, a large cache coherence algorithm, and safety and liveness for a version of Lamport's N-process bakery algorithm for mutual exclusion [35]. McMillan's approach is substantially deductive. The rules of inference, such as symmetry reduction and compositional verification, are specialized but quite powerful.

Seger [46] has extended the Voss tool for symbolic trajectory evaluation with lightweight theorem proving. Symbolic trajectory evaluation (STE) which is a limited form of linear temporal logic model checking. A few simple proof rules are used to decompose proof obligations on the basis of the logical connectives such as conjunction, disjunction, and implication. These rules can be used to decompose a large model checking problem into smaller ones.

2.3 Abstraction and Model Checking

Abstraction has been studied in the context of model checking as a technique for reducing infinite-state or large finite-state models to finite-state models of manageable size [3, 30, 12, 32, 17, 5].

Some of the work on abstraction is based on *data abstraction* where a variable X over a concrete datatype T is mapped to a variable x over an abstract type t. For example, a variable over the natural numbers could be replaced by a boolean variable representing the parity of its value. Clarke, Grumberg, and Long [12] gave a simple criterion for abstractions that preserve $\forall CTL^{*+}$ properties. Let the concrete transition system be given by $\langle I_C, N_C \rangle$ where I_C is the initialization predicate and N_C is the next-state relation. Then the verification of a concrete judgement $\langle I_C, N_C \rangle \models P_C$ can be reduced by means of the abstraction function α to the verification of an abstract judgement $\langle I_A, N_A \rangle \models P_A$ provided

1. $I_C \sqsubseteq I_A \circ \alpha$
2. $N_C \sqsubseteq N_A \circ \langle \alpha, \alpha \rangle$
3. $P_A \circ \alpha \sqsubseteq P_C$

Data abstraction has the advantage that the abstract description can be statically constructed from the concrete program. The drawback is that many useful abstractions are on relations between variables rather than on individual variables.

Graf and Saïdi [44] introduced *predicate abstraction* as a way of replacing predicates or relations over a set of variables by the corresponding boolean variables. For example, given two variables x and y over the integers, and the predicate $x < y$ over these variables, predicate abstraction would replace the variables x and y by a boolean variable b that represents the behavior of the predicate.

The application of predicate abstraction makes significant use of theorem proving. Graf and Saïdi used predicate abstraction to construct an abstract reachability graph for a concrete program by a process of elimination. If a represent an abstract state, a' a putative successor, $\gamma(a)$ the concrete state corresponding to a, and $\gamma(a')$ the concrete state corresponding to a', then if

$$\gamma(a) \supset wp(P)(\neg(\gamma(a')))$$

is provable, the corresponding transition between a and a' can be ruled out.[3] However, if a proof attempt fails, the corresponding successor node can be conservatively included in the abstract reachability graph. Using predicate abstractions with the PVS theorem prover [39], Graf and Saïdi [44] were able to verify a variant of the alternating bit protocol called the bounded retransmission protocol [25]. Das, Dill, and Park [18] extended this technique using the SVC decision procedures [2] and were able to verify such impressive examples as the FLASH cache coherence protocol, and a cooperative garbage collector.

Predicate abstraction can also be used to construct an abstract transition relation instead of the abstract reachability graph. It is typically less expensive to construct the abstract transition relation since fewer proof obligations

[3] All programs are assumed to be total as transition system, i.e., the domain of the next-state relation is the set of all states. Thus, $wp(P)(A)$ is the set of states that have no transitions in P to states in $\neg A$. The dual notion $sp(P)(A)$ is the set of states reachable from some state in A by a transition of P.

are generated, but it typically results in a coarser abstraction than one that is obtained by directly computing the abstract reachability graph. In the latter construction, information about the current set of abstract reachable states can be used to rule out unreachable successor states. Bensalem, Lakhnech, and Owre [5] describe an abstraction tool called InVeSt that uses the elimination method to construct an abstract transition system from a concrete one in a compositional manner. Colon and Uribe [13] give another compositional method for constructing abstractions with the framework of the STeP theorem prover [33].

All of the above abstraction techniques preserve only $\forall CTL^{*+}$ properties, namely those in the positive fragment of CTL^* with universal path quantification. For more general calculi, criteria for abstractions that preserve CTL* [16] and mu-calculus [32], but these results are quite technical. Saïdi and Shankar [45] gave a simple method for constructing predicate abstractions over the full relational mu-calculus [40]. The two key observations in this work are:

1. The operators of the mu-calculus are monotonic with respect to upper and lower approximations.

2. The over-approximation of a literal (an atomic formula or its negation) can be efficiently computed in conjunctive normal form by using a theorem prover as an oracle.

Verification diagrams [34] can also be seen as a form of predicate abstraction. These diagrams employ graphs whose nodes are labeled by assertions and the edges correspond to program transitions within the diagram. Properties can be directly checked with respect to the verification diagram.

The primary advantage of predicate abstraction is that it is sufficient to guess a relevant predicates without having to guess the exact invariant in these predicates. For n predicates, the construction of the abstract transition system generates of the order of 2^n proof obligations. The resulting abstract model can also be model checked in time that is exponential in n to yield useful invariants. With deduction, there are 2^{2^n} boolean functions that are candidate invariants in these n predicates so that it is harder to guess suitable invariants.

2.4 Automatic Invariant Generation

Automatic invariant generation has been studied since the 1970s [15, 21, 29, 48]. This study has recently been revived through the work of Bjørner, Browne, and Manna [8], and Bensalem, Lakhnech, and Saïdi [6, 43, 4].

The strongest invariant of a transition system P is given by the least fixed point starting from the initial states of P of the strongest postcondition operator for P, $\mu X.I_P \vee sp(P)(X)$. If this computation terminates, it would yield the set of reachable states of P which is its strongest invariant. Unfortunately, the least fixed point computation rarely terminates for infinite-state systems. A program with a single integer variable x that is initially 0 and is repeatedly incremented by one, yields a nonterminating least fixed point computation.

Widening techniques [14] are needed to accelerate the fixed point computation so that it does terminate with a fixed point that is not necessarily the least one.

A different, more conservative approach to invariant generation is given by the computation of the greatest fixed point of the strongest postcondition $\nu X.sp(P)(X)$. For example, a greatest fixed point computation on a program with a single variable x and a single guarded transition $x \geq 0 \longrightarrow x := x + 1$ would terminate and yield the invariant $x \geq 0$. The greatest fixed point invariant computation also may not terminate and could require *narrowing* as a way of accelerating termination. However, one could stop the greatest fixed point computation after any bounded number of iterations and the resulting predicate would always be a valid invariant.

Dually, a putative invariant p can be strengthened to an inductive one by computing the greatest fixed point with respect to the weakest precondition of the program of the given invariant $\nu X.p \wedge wp(P)(X)$. If this computation terminates, the result is an invariant that is inductive.

Automatic invariant generation is not yet a successful technology. Right now, it is best used for propagating invariants that are computed from other sources by taking the greatest fixed point with respect to the strongest post-condition starting from a known invariant. However, as theorem proving technology becomes more powerful and efficient, invariant generation is likely to be quite a fruitful technique.

3 Symbolic Analysis

Symbolic analysis is simply the computation of fixed point properties of programs through a combination of deductive and explorative techniques. We have already seen the key elements of symbolic analysis as

1. *Automated deduction*, in computing property preserving abstractions and propagating the consequences of known properties.

2. *Model checking*, as a means of computing global properties of by means of systematic symbolic exploration. For this purpose, model checking is used for actually computing fixed points such as the reachable state set, in addition to verifying given temporal properties.

3. *Invariant generation*, as a technique for computing useful properties and propagating known properties.

3.1 SAL: A Symbolic Analysis Laboratory

SAL is a framework for integrating different symbolic analysis techniques including theorem proving and model checking. The core of SAL is a description language for transition systems. The design of this intermediate language has been influenced by SMV [37], UNITY [10], Murphi [38], and Reactive Modules [1]. Transition systems described in SAL consist of modules with input,

output, global, and local variables. Initializations and transitions can be either specified by definitions of the form *variable* = *expression* or by guarded commands. The assignment part of a guarded command consists of assignments of the form $x' = \textit{expression}$, meaning the new value of x is the value of the *expression*, as well as selections $x' \in \textit{set}$, meaning the new value of x is nondeterministically selected from the value of the nonempty set *set*. SAL is a synchronous language in the spirit of Esterel [7], Lustre [23], and Reactive Modules [1], in the sense that transitions can depend on latched values as well as current inputs. SAL modules can be composed by means of

1. Binary synchronous composition $P||Q$ whose transitions consist of lockstep parallel transitions of P and Q.
2. Binary asynchronous composition $P\,[]\,Q$ whose transitions are the interleaving of those of P and Q.
3. N-fold synchronous composition $(||\ (i) :\ P[i])$
4. N-fold asynchronous composition $([]\ (i) :\ P[i])$

The implementation of SAL is still ongoing. The version to be released some time in 2000 will consist of a parser, typechecker, translators to SMV and PVS, a translator to Java (for animation), and a translator from Verilog, among other tools.

Since the SAL implementation is still incomplete, we informally describe some examples that motivate the need for a symbolic analysis framework integrating abstraction, invariant generation, theorem proving, and model checking.

3.2 Analysis of a Two Process Mutual Exclusion Algorithm

As a first example, we use a simplified 2-process version of Lamport's Bakery algorithm for mutual exclusion [31]. The algorithm consists of two processes P and Q with control variables *pcp* and *pcq*, respectively, and shared variables x and y. The control states of these processes are either `sleeping`, `trying`, or `critical`. Initially, *pcp* and *pcq* are both set to `sleeping` and the control variables satisfy $x = y = 0$. The transitions for P are

$$\begin{array}{rrl} & \texttt{pcp} = \textit{sleeping} & \longrightarrow\ x' = y + 1;\ \textit{pcp}' = \texttt{trying} \\ [] & \textit{pcp} = \texttt{trying} \wedge (y = 0 \vee x < y) & \longrightarrow\ \textit{pcp}' = \texttt{critical} \\ [] & \textit{pcp} = \texttt{critical} & \longrightarrow\ x' = 0;\ \texttt{pcp}' = \texttt{sleeping} \end{array}$$

Similarly, the transitions for Q are

$$\begin{array}{rrl} & \textit{pcq} = \texttt{sleeping} & \longrightarrow\ y' = x + 1;\ \textit{pcq}' = \texttt{trying} \\ [] & \textit{pcq} = \texttt{trying} \wedge (x = 0 \vee y \leq x) & \longrightarrow\ \textit{pc}' = \texttt{critical} \\ [] & \textit{pcq} = \texttt{critical} & \longrightarrow\ y' = 0;\ \textit{pcq}' = \texttt{sleeping} \end{array}$$

The invariant we wish to establish for $P \,[]\, Q$ is $\neg(pcp = \texttt{critical} \wedge pcq = \texttt{critical}$. Note that $P \,[]\, Q$ is an infinite-state system and in fact the values of the variables x and y can increase without bound. We can therefore attempt to verify the invariant by means of a property-preserving predicate abstraction to a finite-state system.

The abstraction predicates suggest themselves from the initializations, guards, and assignments. We therefore abstract the predicate $x = 0$ with the boolean variable x_0, the predicate $y = 0$ with the boolean variable y_0, and the predicate $x < y$ with the boolean variable xy. The resulting abstract system can be computed as P' and Q', where in the initial state, $x_0 \wedge y_0 \wedge \neg xy$, and the transitions for P' are

$$\begin{array}{rrl} & pcp = \textit{sleeping} \longrightarrow & \mathbf{x_0' = false};\ xy' = \mathbf{false};\ pcp' = \texttt{trying}; \\ [] & pcp = \texttt{trying} \wedge (y_0 \vee xy) \longrightarrow & pcp' = \texttt{critical}; \\ [] & pcp = \texttt{critical} \longrightarrow & x_0' = \mathbf{true}; xy' \in \{\mathbf{true}, \mathbf{false}\};\ pcp' = \texttt{sleeping}; \end{array}$$

The transitions for Q' are

$$\begin{array}{rrl} & pcq = \texttt{sleeping} \longrightarrow & y_0' = \mathbf{false};\ xy' = \mathbf{true};\ pcq' = \texttt{trying}; \\ [] & pcq = \texttt{trying} \wedge (x_0 \vee \neg xy) \longrightarrow & pcp' = \texttt{critical}; \\ [] & pcp = \texttt{critical} \longrightarrow & y_0' = \mathbf{true}; xy' = \mathbf{false};\ pcp' = \texttt{sleeping}; \end{array}$$

Model checking the abstract system $P' \,[]\, Q'$ easily verifies the invariant

$$\neg(pcp = \texttt{critical} \wedge pcq = \texttt{critical}).$$

The theorem proving needed to construct the abstraction is at a trivial level that can be handled automatically by the decision procedures over quantifier-free formulas in a combination of theories [42]. Such decision procedures are present in systems like PVS [39], ESC [19], SVC [2], and STeP [33]. The above example can be verified fully automatically by means of the `abstract-and-model-check` command in PVS [45].

3.3 Analysis of an N-Process Mutual Exclusion Algorithm

We next examine a fictional example, namely, one that has not been mechanically verified by us. This example is a simplified form of the N-process Bakery algorithm due to Lamport [31]. The description below shows a hand-executed symbolic analysis.

In this version of the Bakery algorithm, there are N processes $P(0)$ to $P(N-1)$, with a shared array x of size N over the natural numbers. The logical variables i, j, and k range over the subrange $0..(N-1)$. The operation $max(x)$ returns the maximal element in the array x. Initially, each $P(i)$ is in the control state `sleeping`, and for each i, $x(i) = 0$. Let $\langle x, i\rangle \leq \langle y, j\rangle$ be defined as the lexicographic ordering $x < y \vee (x = y \wedge i \leq j)$. We abbreviate $y = 0 \vee \langle x, i\rangle \leq \langle y, j\rangle$ as $\langle x, i\rangle \preceq \langle y, j\rangle$.

The transitions of processes $P(i)$ for $0 \leq i < N$ are interleaved and each non-stuttering transition executes one of the following guarded commands.

$$
\begin{array}{rll}
 & pc(i) = \mathit{sleeping} \longrightarrow & x'(i) = 1 + \mathit{max}(x); \\
 & & pc'(i) = \texttt{trying}; \\
[] \quad pc(i) = \texttt{trying} & \longrightarrow & pc'(i) = \texttt{critical}; \\
\wedge \quad (\forall j : \langle x(i), i\rangle \preceq \langle x(j), j\rangle) & & \\
[] \quad pc(i) = \mathit{critical} & \longrightarrow & x'(i) = 0; \\
 & & pc'(i) = \texttt{sleeping};
\end{array}
$$

We want to prove the invariance property

$$(\forall i : pc(i) = \texttt{critical} \supset (\forall j : pc(j) = \texttt{critical} \supset i = j)). \tag{1}$$

Invariant generation techniques can be used to generate trivial invariants such as

$$(\forall i : x(i) = 0 \ \mathit{iff}\ pc(i) = \texttt{sleeping}). \tag{2}$$

We omit the details of the invariant generation step. The above invariant will prove useful in the next stage of the analysis.

We next skolemize the mutual exclusion statement so as to obtain a correctness goal about a specific but arbitrary i which we call a. The main invariant now becomes

$$pc(a) = \texttt{critical} \supset (\forall j : pc(j) = \texttt{critical} \supset a = j) \tag{3}$$

The goal now is to reduce the N-process protocol to a two process protocol consisting of process a and another process b that is an *existential abstraction* of the remaining $N-1$ processes. By an existential abstraction, we mean one where the $N-1$ processes are represented by a single process b such that a transition by any of the $N-1$ processes is mapped to a corresponding transition of b. In such an abstraction, b is in control state `critical` if any one of the $N-1$ processes is critical. Otherwise, b is in control state `trying` if none of the $N-1$ processes is in the state `critical` and at least one of them is in its `trying` state. If none of the $N-1$ process is either `trying` or `critical`, then b is in its `sleeping` state.

By examining the predicates appearing in the initialization, guards, and the property, we can directly obtain the following abstraction predicates given by the function γ which maps abstract variables to the corresponding concrete predicates:

$$
\begin{array}{lcll}
\gamma(pca) & = & pc(a) & \\
\gamma(pcb) & = & \mathit{if} & (\exists j : j \neq a \wedge pc(j) = \texttt{critical}) \\
 & & \mathit{then} & \texttt{critical} \\
 & & \mathit{elsif} & (\exists j : j \neq a \wedge pc(j) = \texttt{trying}) \\
 & & \mathit{then} & \texttt{trying} \\
 & & \mathit{else} & \texttt{sleeping} \\
\gamma(xa_0) & = & (x(a) = 0) &
\end{array}
$$

$$
\begin{array}{rll}
 & pca = \texttt{sleeping} \longrightarrow & xa' = \textbf{false}; \\
 & & ma' = xb; \\
 & & pca' = \texttt{trying}; \\
[] & pca = \mathit{trying} \wedge ma \longrightarrow & pca' = \texttt{critical}; \\
[] & pca = \texttt{critical} \longrightarrow & pca' = \texttt{sleeping}; \\
 & & ma' = xb; \\
 & & ea' = \neg(pcb = \texttt{critical}); \\
 & & xa' = \textbf{true}; \\
[] & pcb = \texttt{sleeping} \longrightarrow & pcb' = \texttt{trying};\ xb' = \textbf{false}; \\
 & & ma' = \neg xa \\
[] & pcb = \texttt{trying} \wedge \neg ma \longrightarrow & pcb' = \texttt{critical};\ ea' = \textbf{false}; \\
[] & pcb = \texttt{critical} \longrightarrow & pcb' = \texttt{sleeping}; \\
 & & ea' = \textbf{true}; \\
 & & ma' = \textbf{true}; \\
 & & xb' = \textbf{true}; \\
[] & pcb = \texttt{critical} \longrightarrow & pcb' = \texttt{trying}; \\
 & & ea' = \mathit{true}; \\
 & & ma' \in \{\textbf{true}, ma\}; \\
[] & pcb = \texttt{critical} \longrightarrow & ma' \in \{\textbf{true}, ma\};
\end{array}
$$

Figure 1: Abstract transitions for the N-process Bakery Algorithm

$$
\begin{array}{rcl}
\gamma(xb_0) & = & (\forall j : j \neq a \supset x(j) = 0) \\
\gamma(ma) & = & (\forall j : \langle x(a), a\rangle \preceq \langle x(j), j\rangle) \\
\gamma(mb) & = & (\exists j : (\forall k : \langle x(j), j\rangle \preceq \langle x(k), k\rangle) \\
\gamma(ea) & = & (\forall j : pc(j) = \texttt{critical} \supset a = j)
\end{array}
$$

Since mb is only relevant when $pc(j) = \mathit{trying}$ for $j \neq a$, we can use invariant (2) to prove that

$$j \neq a \wedge pc(j) \neq \mathit{sleeping} \supset \gamma(mb) = \gamma(\neg ma)$$

thereby dispensing with mb in the abstraction.

With the above abstraction mapping, the goal invariant (3) becomes

$$pca = \texttt{critical} \supset ea.$$

and the resulting abstracted transition system is one where initially

$$pca = \mathit{sleeping} \wedge pcb = \mathit{sleeping} \wedge xa_0 \wedge xb_0 \wedge ma \wedge ea$$

Each non-stuttering step in the computation of the abstract program executes one of the guarded commands shown in Figure 1.

Model checking the abstract protocol fails to verify the invariant

$$pca = \mathit{critical} \supset ea$$

as the model checker could generate the following counterexample sequence of transitions:

transition	*pca*	*xa*	*ma*	*ea*	*pcb*	*xb*
initially	*sleeping*	*true*	*true*	*true*	*sleeping*	*true*
3	*sleeping*	*true*	*false*	*true*	*trying*	*false*
4	*sleeping*	*true*	*false*	*false*	*critical*	*false*
1	*trying*	*false*	*false*	*false*	*critical*	*false*
8	*trying*	*false*	*true*	*false*	*critical*	*false*
2	*critical*	*false*	*true*	*false*	*critical*	*false*

An inspection of the counterexample and the abstract model confirms that the mutual exclusion invariant would follow if the invariant $\neg xa \wedge ma \supset ea$ were to hold. Mapped back in the concrete domain, this corresponds to

$$\forall i : x(i) \neq 0 \wedge (\forall j : x(j) = 0 \vee \langle x(i), i\rangle \leq \langle x(j), j\rangle) \supset (\forall j : pc(j) = \texttt{critical} \supset i = j).$$

This goal can be generalized as

$$(\forall i, j : x(i) \neq 0 \wedge (x(j) = 0 \vee \langle x(i), i\rangle) \leq \langle x(j), j\rangle \supset (pc(j) = \texttt{critical} \supset i = j)).$$

and further rearranged as

$$(\forall i, j : pc(j) = \texttt{critical} \supset (x(i) \neq 0 \wedge (x(j) = 0 \vee \langle x(i), i\rangle \leq \langle x(j), j\rangle))) \supset i = j).$$

By the invariant (2), we can eliminate the subformula $x(j) = 0$ and simplify the goal to the equivalent formula

$$(\forall i, j : pc(j) = critical \supset x(i) = 0 \vee \langle x(j), j\rangle \leq \langle x(i), i\rangle).$$

This can be rearranged as

$$(\forall j : pc(j) = critical \supset (\forall i : x(i) = 0 \vee \langle x(j), j\rangle \leq \langle x(i), i\rangle))).$$

But this is the just the invariant $pca = critical \supset ma$ which is already implied by the abstract model.

The safety property is thus verified by using a judicious combination of a small amount of theorem proving and model checking. The abstractions were suggested by the predicates in the text of the program. Simple invariant generation methods were adequate for generating trivial invariants. Theorem proving in the context of these invariants could be used to discharge the proof obligations needed to construct an accurate abstraction of the N-process protocol. Abstraction mappings of this sort are quite standard and work for many mutual exclusion and cache consistency algorithms [47]. The abstract model did not discharge the main safety invariant but it was easy to extract the minimal condition needed to verify the invariant from the abstract model. A reachability analysis of the abstract model delivered enough useful invariants so that a small amount of theorem proving could discharge this condition. Neither the model checking nor the theorem proving used here is especially difficult. While some guidance is needed in selecting lemmas and conjectures, the proofs of these can be carried out with substantial automation.

4 Conclusion

We have argued that verification technology is best employed as an analysis technique to generate properties of specifications and programs rather than as a method for establishing the correctness of specific properties. Such a symbolic analysis framework can employ both theorem proving and model checking as appropriate to generate useful abstractions and automatically derive system properties.

Many ideas remain to be explored within the symbolic analysis framework. The construction of the symbolic analysis laboratory SAL as an open framework will support the exploration of ideas at the interface of theorem proving and model checking.

Acknowledgments. Many collaborators and colleagues have contributed ideas and code to the SAL language and framework, including Saddek Bensalem, David Dill, Tom Henzinger, Luca de Alfaro, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Eli Singerman, Mandayam Srivas, Jens Skakkebæk, and Ashish Tiwari. John Rushby read an earlier draft of the paper and suggested numerous improvements.

References

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, IEEE Computer Society Press, New Brunswick, New Jersey, 27–30 July 1996.

[2] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, Volume 1166 of Springer-Verlag *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996.

[3] Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property preserving simulations. In *Computer-Aided Verification, CAV '92*, Volume 630 of Springer-Verlag *Lecture Notes in Computer Science*, pages 260–273, Montréal, Canada, June 1992. Extended version available with title "Property Preserving Abstractions.".

[4] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.

[5] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [26], pages 319–331.

[6] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/August 1996.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[11] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[12] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[13] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [26], pages 293–304.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis. In *4th ACM Symposium on Principles of Programming Languages.* January 1977.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages.* January 1978.

[16] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 561–581, 1994.

[17] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996.

[18] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [24], pages 160–171.

[19] David L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, Association for Computing Machinery, San Diego, CA, January 1996.

[20] Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Computer-Aided Verification, CAV ' 95*, 1995. This volume.

[21] S. M. German and B. Wegbreit. A synthesizer for inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[22] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[24] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[25] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, March 1994.

[26] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998.

[27] G. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[28] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*. 1993.

[29] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.

[30] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1994.

[31] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[32] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[33] Z. Manna and the STeP Group. STeP: The Stanford Temporal Prover. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, Volume 915 of Springer-Verlag *Lecture Notes in Computer Science*, pages 793–794, Aarhus, Denmark, May 1995.

[34] Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In Armando M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, Volume 1548 of Springer-Verlag *Lecture Notes in Computer Science*, pages 28–41, Amazonia, Brazil, January 1999.

[35] K. McMillan, S. Qadeer, and J. Saxe. Induction in compositional model checking. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification*, Volume 1855 of Springer-Verlag *Lecture Notes in Computer Science*, pages 312–327, Chicago, IL, July 2000.

[36] K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods.* pages 219–233, September 1999.

[37] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[38] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual.* Computer Science Department, Stanford University, Stanford, CA, March 1993.

[39] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Artifical Intelligence.* pages 748–752, June 1992.

[40] David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[41] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, Volume 939 of Springer-Verlag *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995.

[42] H. Rueß and N. Shankar. Deconstructing Shostak. Available from `http://www.csl.sri.com/shankar/shostak2000.ps.gz`., January 2000.

[43] Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, Volume 1055 of Springer-Verlag *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996.

[44] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[45] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [24], pages 443–454.

[46] Carl-Johan H. Seger. Formal methods in CAD from an industrial perspective. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, Volume 1522 of Springer-Verlag *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998.

[47] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Science*, pages 499–528. Springer, 1997.

[48] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.

Part V

Technology Transition

Overview

Key results generated by this project have had a significant impact on the computer system development process at Rockwell Collins. For example, the encapsulation mechanisms developed under this DARPA contract have influenced both the design philosophy and the design process at Rockwell Collins, and are reflected in their partitioning architecture for both civil and military avionics. The hallmarks of this architecture, namely simple memory management hardware, guaranteed time slicing, and mediated interrupt handling, constitute the core of Rockwell Collins' common computing platform for avionics. The notion of invariant performance, another significant contribution of this project, has become an essential component of the basic contract between the developers and users of the new Rockwell Collins partitioning environment [5]. The further utility of this notion for technology transition derives from the cogent claim that invariant performance provides the strongest possible composability argument with respect to system certification [4].

This project has also yielded methods for automated modeling and reasoning that are being integrated into the Rockwell Collins computer system development cycle. These methods include the use of symbolic execution to explore design choices, and the use of symbolic results to enhance regression testing [3]. In the longer term, Rockwell Collins expects executable formal models to become an integral part of their processor development process [18].

Invariant Performance and PMU Design Considerations

David Greve Matthew Wilding
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA 52498 USA

Abstract

This report outlines some of the design considerations surrounding the development of the JEM2 PMU and documents the impact the concept of invariant performance has had on the PMU implementation.

1 Introduction

Digital flight-control functions in modern aircraft are typically implemented using a federated architecture. A federated architecture is one in which each function has its own computer system that is only loosely coupled to the computer systems of other functions. Typically, functions in a federated architecture are physically separated and dependencies between functions are limited to the exchange of sensor and control data. Physical separation and limited functional dependency provide strong functional isolation in federated systems. As a result, a fault or error in the computer hardware or software implementing one function is unlikely to propagate to other functions. It is this isolation that allows for independent certification of functionally distinct systems.

Integrated Modular Avionics (IMA) has received significant attention as an alternative to the federated architecture. In IMA a single computer system provides a common computing resource for several functions. Centralizing this functionality reduces the resource requirements of an avionics suite while capitalizing on the computational capabilities of modern computing systems. Crucial to the success of IMA, however, is the concept of composability. Composability allows the various functions hosted by the IMA computer system to be certified once, independently, and only to a level appropriate to the criticality of the function. Ideally, each function then retains its certification when composed with other functions on the same IMA system.

Note that the IMA model of composability is identical to what one finds in a federated system in which each function is hosted on a dedicated computer system: the functions are certified once, independently, and only to a level appropriate to the criticality of the function they perform. With this observation it is clear that anything less than full composability will significantly increase the cost of certifying a given avionics suite in an IMA system. Because certification is a major portion of the cost of an avionics system, any savings in hardware costs will rapidly evaporate in the absence of composability.

The fact that the IMA computer system is a shared resource, however, is in direct conflict with the concept of isolation so carefully maintained in the federated architecture. Isolation, such as one finds in a federated architecture, is the key to composability. In order to counteract the lack of physical separation and to achieve the degree of functional isolation required to support composability, an IMA computer system must employ partitioning.

Partitioning is a technique for isolating functions in an effort to extend the current federated architecture certification process to cover integrated systems. Partitioned systems provide isolation in space through memory protection and in time through periodic partition switching. Integrated systems then rely on this spatio-temporal isolation of functions executing on the same computer system to emulate the logical isolation occurring naturally in a federated system as a result of physical isolation.

Invariant performance is a concept that was developed to describe the properties of an ideal partitioned system[3]. The invariant performance property states that, given a partition schedule, all aspects of the operation of a given

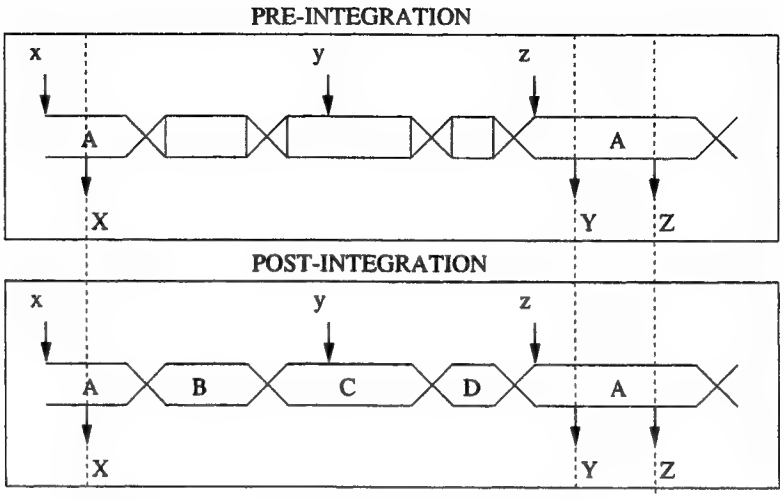


Figure 1: Invariant Performance Illustration

partition are strict functions of the logical state of the partition and its inputs. Invariant performance guarantees that, given identical inputs, the outputs and the execution of a partition are identical regardless of the activity of any other partition in the system.

Invariant performance provides the developer with a contract that states that any function developed in a partitioned environment will operate identically before and after system integration. It also provides the certification authority with the assurance that any test run on a function in a partitioned system will have exactly the same results following system integration. In this sense, invariant performance provides the strongest possible argument for the composability of integrated modular avionics systems.

Figure 1 illustrates some of the aspects of invariant performance. The upper time line in this figure shows that partition A is scheduled to run during a particular time slot and that it receives inputs (x, y, and z) and produces output values (X, Y, and Z) with a particular temporal relationship. The bottom time shows that, following system integration, if partition A is given the same input values at the same relative time, it produces the same output values and they are available at the same relative time.

2 JEM2/PMU Design Objectives

The JEM2/PMU is designed to support embedded real-time safety-critical partitioned systems that are capable of exhibiting invariant performance. We describe several aspects of this design, focusing particularly on how the invariant performance concept is reflected in the development of this proprietary device.

The first three design objectives of the JEM2/PMU are addressed directly by the fundamental architecture of the JEM[5]. The JEM is a descendant of a long line of stack-based processors designed by Rockwell with similar objectives[1]. The high code density, small die size, and low power consumption of the JEM processor family makes it ideal for embedded applications. Likewise, microcode support for interrupt handling and thread scheduling make it attractive in real-time systems. The direct support for object oriented programming, in the form of virtual functions and data objects, as well as the strong type system encourage good programming style and help in developing safety-critical applications.

The goal of partitioning, however, is somewhat contrary to the first three system objectives. Sophisticated memory management to support spatial isolation of functions can lead to increased cost and reduced performance, contrary to the embedded system and real-time design goals. The need provide temporal isolation between the various partitions can infringe on the real-time behavior. Finally, the complexity of many partitioning schemes threatens the very level of safety they set out to secure in the first place. The JEM2/PMU design strategy addresses these issues through the use of a virtual machine partitioning scheme supporting high rate context switching.

2.1 Virtual Machine Partitioning

There are two common models that might be followed when developing a partitioning system: a tasking model or a virtual machine model[2]. Figure 2 illustrates these two models. In the virtual machine model, both the applications and the operating system are isolated from the partitioning kernel. The partitioning kernel performs only the bare essential tasks of system initialization, partition configuration and partition scheduling. Each partition, therefore, has its own operating system to allocate resources and perform thread scheduling; behaving effectively like an independent machine. In the tasking model, the partitioning kernel and the operating system are essentially one and the same. In this model, partitioning is performed implicitly at the task level as a by-product of the thread scheduling process and each partition shares the various resources offered by the host operating system.

Under invariant performance, the behavior of one partition, no matter how corrupt or malicious, must not affect the performance of another partition. Given this requirement, it was decided that the partitioning scheme that afforded the greatest degree of protection would be a virtual machine partitioning scheme. This choice proved advantageous in a number of ways.

2.1.1 Straightforward Memory Protection

Because of the course-grain nature of virtual machine partitioning, the memory protection logic can be quite simple, allowing for the development of simple, high-performance memory protection circuitry. The system level separation of the memory into kernel-specific and application-specific memory protects against the mixing of critical and non-critical memory spaces and helps enforce

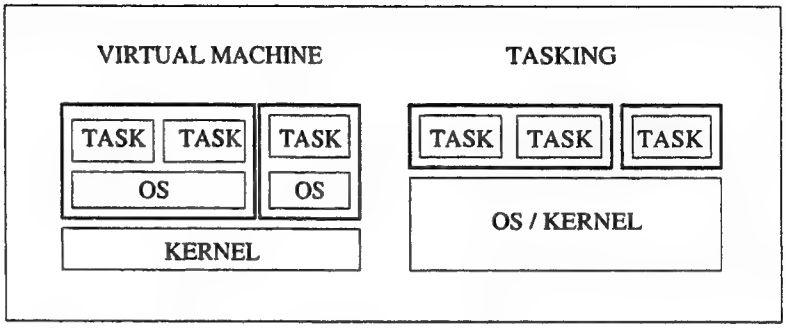


Figure 2: Partitioning Models

separation of crucial data structures. Using this technique, one can predefine the programming data for the memory protection logic in the protected memory space accessible only by the partitioning kernel and this data can be saved in ROM. Thus, there is no dependence on run-time code for memory allocation and it is possible to perform static analysis of the memory protection properties based only on the data written to ROM during system programming.

2.1.2 High Rate Context Switching

In real-time systems, interrupt latency is a crucial factor. In partitioned systems, however, a trade-off exists between interrupt latency and time determinism. Using an immediate interrupt service model, where any interrupt will suspend the currently execution partition to allow interrupt handler code to execute, the system interrupt latency can be low. However, without a sophisticated partition timing scheme it is virtually impossible to provide deterministic partition execution. For that matter, without governors on external interrupts, it is difficult even to provide partition execution bounds.

A more deterministic alternative, however, which is to wait until the end of the currently scheduled partition to service interrupts, bounds the minimum interrupt latency below by the maximum partition time slice. This is of particular concern for systems in which partition context switching overhead is significant. The JEM2/PMU opts in favor of deterministic execution (invariant performance) over interrupt latency, but then makes a concerted effort to minimize partition context switching overhead.

In general, partition context switching involves suspending a current partition and dispatching another. Suspending a partition involves saving the crucial state of that partition so that execution can later resume where it left off. Dispatching a partition involves programming the memory and time management

utilities of the partition management unit, restoring partition state and then resuming partition execution.

There are essentially two aspects of the JEM2/PMU system that enable support of high rate context switching: minimal system state and configuration caching. The JEM family of processors has a relatively small internal state. This simple fact enables rapid saving and restoring of context, an important feature in in real-time systems. Reprogramming the memory protection logic of the PMU, however, can be a lengthy process involving writing many different configuration words to several different registers. To address this issue the PMU supports configuration caching. In a nutshell, configuration caching allows the processor to write an identifying partition number into a PMU configuration register to instantly activate a preprogrammed set of memory protection logic. This combination of limited processor state and PMU configuration caching allows the JEM2/PMU to support the very fast partition switch rates needed for realistic real-time systems.

2.1.3 Implementation Complexity

The JEM2/PMU partitioning scheme relies on specialized, privileged microcode in the JEM2 to perform initialization, partition configuration and partition scheduling. This microcode represents the entirety of the JEM partitioning system: it is not possible to execute software in privileged kernel mode. This limitation serves to keep the issue of partition kernel verification tractable. Because the tasks performed by the partitioning kernel are simple, its implementation can be kept small. Ideally, the handful of microcode used to perform this task can then be subjected to rigorous formal analysis to guarantee correctness under all operating conditions.

2.2 Invariant Performance

The development of the concept of invariant performance has had a significant impact on the design decisions made in the development of the Rockwell Collins JEM2/PMU. One of the design requirements of the JEM2/PMU is that it provide the strongest possible incremental certification story to support the requirements of integrated modular avionics systems. The operating assumption is that anything less than full composability will significantly increase the cost of certifying avionics suites, potentially overshadowing any savings obtained from reduced component costs. For these reasons, invariant performance has been considered a priority throughout the design process.

In order to obtain invariant performance in a partitioned system, certain restrictions must be met. These include, among others, a completely specified processor with bounded interrupt latency whose privileged mode of operation is well contained, a memory protection unit that can prohibit memory transactions outside of the predefined bounds of a partition, a partition timer that can end of one partition time interval, and a "MUCOS" timer[3] that can deterministically

mark the beginning of subsequent partition execution. All of these issues are addressed and solved in the JEM2/PMU design.

2.3 Additional PMU Features

In addition to the partition timer, MUCOS timer and configuration cache the PMU also provides other partition-sensitive services to the system. An interesting result of choosing a virtual machine partitioning scheme over a threaded scheme is that operating system resources must now be duplicated within the context of each virtual machine. This includes such mechanisms as interrupt controllers and timers. As a result the PMU also provides a set of runtime resources for each supported partitions. These resources include a global clock, runtime counters for delay and cyclic interrupts, a dedicated interrupt controller, and an interrupt mailbox and subscription service for sending and receiving interrupts between partitions.

3 Compliance Proof Strategy

A crucial aspect of invariant performance is that the processor itself cannot be used to violate partitioning. In particular, it is important that the processor have a protected (or user) mode, in which memory transactions can be checked against predefined acceptable ranges, and a privileged (or kernel) mode in which the processor can access the partitioning data structures as well as reconfigure the memory management hardware. It is crucial that user tasks cannot cause the processor to perform privileged memory transactions and it is important that entry into and exit from the privileged mode be well defined and guaranteed to be consistent with invariant performance. Interestingly, it is not important that any of the user mode processor instructions work correctly in order to satisfy invariant performance; only that they do not result in protected transactions or an unexpected transition into protected mode.

Timing issues are also a concern in partitioned systems. For example, the processor must have an acceptably tight bound on interrupt latency. In particular, there can be no circumstances under which the processor fails to respond to a partition interrupt. Demonstrating this property involves identifying the interrupt response sequence and showing that, regardless of the current state, the longest sequence of events leading to an interrupt response has an acceptable bound.

Uniform start times are also crucial to invariant performance. Implementation of uniform start times requires that the system have the ability to stall the processor while waiting for the MUCOS timer to time out. Such stalling might be implemented with a wait-for-interrupt style instruction or, as we have done, using a delayed-acknowledge bus arbitration scheme. The time it takes for the processor to resume execution following the MUCOS time-out must be fixed and it must be impossible to get into a stalled state except in response to a partition interrupt or an early partition exit.

To support these verification activities, an Executable Formal Model (EFM) of the JEM2 and the JEM2 PMU have been constructed. EFMs allow for the generation of a functional simulator of a hardware device directly from the formal model[4]. As a result, in addition to enabling formal reasoning about models of hardware devices, this process supports the validation of the formal models through their use as a simulator in an engineering design environment.

While development of such formal models is straightforward, the process of constructing a simulator for a new hardware device is relatively complex. As a result, while models of the entire JEM2 and the JEM2 PMU have been constructed, no substantial correctness proofs have yet been attempted. The JEM2 model, however, has been validated through the execution of test programs and the PMU model has been integrated with a VHDL simulation environment to enable its use in a PMU test bench circuit.

4 Conclusion

This report discussed the design objectives for a real partitioning system, specifically the design objectives for the JEM2/PMU system, showed how invariant performance impacted the design decisions associated with that system, and demonstrated that it is possible to construct realistic systems that exhibit invariant performance.

References

[1] David A. Greve and Matthew M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, January 12, 1998.

[2] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1998. Unpublished draft report; available at http://www.csl.sri.com/ rushby/partitioning.html.

[3] Matthew Wilding, David Hardin, and David Greve. Invariant performance: A statement of task isolation useful for embedded application integration. *Proceedings of Dependable Computing for Critical Applications - DCCA-7*, 1999.

[4] Matthew M. Wilding, David A. Greve, and David S. Hardin. Efficient simulation of formal processor models. Technical report, Rockwell Collins Advanced Technology Center, October 1998. unpublished manuscript.

[5] Alexander Wolfe. First Java-specific MPU rolls. *Electronic Engineering Times*, page 1, September 22, 1997.

Invariant Performance and Commercial System Components*

David Greve
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA 52498 USA

*This work was supported at Rockwell Collins Inc. by the Defense Advanced Research Projects Agency, DARPA order D855. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Rockwell Collins, the Defense Advanced Research Projects Agency, or the US government.

Abstract

In this paper we explore the relationship between invariant performance and the use of commercial system components in certified systems. Invariant performance is a concept developed under DARPA contract D855. This paper describes invariant performance in the context of a partitioned system and explains how it provides the strongest possible composability argument from the perspective of certification. Illustrations highlight the various invariant performance properties and relaxations of those properties are evaluated with respect to their impact on the certification process. The relaxations we consider are consistent with those encountered when using commercial components and software to implement a partitioned system.

1 Introduction

In this paper we explore the relationship between invariant performance and system certification by introducing the concept of invariant performance in a partitioned system and explaining how it provides the strongest possible composability argument from the perspective of certification. We begin by providing motivating factors for considering partitioning in avionics systems as well as introducing much of the terminology used throughout the paper. Some of the subject matter in this paper is taken without attribution from [7, 5, 6].

1.1 Federated Architecture

Digital flight-control functions in modern aircraft are typically implemented using a federated architecture. A federated architecture is one in which each function has its own computer system that is only loosely coupled to the computer systems of other functions. Typically, functions in a federated architecture are physically separated and dependencies between functions are limited to the exchange of sensor and control data. Examples of such functions include autopilot, flight management, and displays.

It should be noted that the physical separation and limited functional dependencies characteristic of federated architectures provide strong functional isolation. That is to say, a fault or error in the computer hardware or software implementing one function is unlikely to propagate to other functions. It is this isolation that provides for independent certification of functionally distinct systems.

Between the various components of an avionics function, however, interactions are not ruled out. The certification of such functions may involve the incremental verification of their various components, from the hardware and peripherals used to construct the system to the operating system and applications running on the hardware. Nonetheless, for the purposes of certification, all of these components must be considered together as a system and all must be verified to the same level of criticality. It is widely understood that one does not certify an individual functional component, one certifies an entire avionics system.

1.2 Integrated Modular Avionics

Integrated Modular Avionics (IMA) has received significant attention as an alternative to the federated architecture. In IMA a single computer system provides a common computing resource to several functions. Centralizing this functionality reduces the resource requirements of an avionics suite while capitalizing on the enhanced computational capabilities of modern computing systems.

Crucial to the success of IMA, however, is the concept of composability. The concept is that the various functions hosted by the IMA computer system are certified once, independently, and only to a level appropriate to the criticality of

the function they perform. Ideally each function will then retain its certification even when composed with other functions on the same IMA system.

Note that the IMA model of composability is identical to what one finds in a federated system in which each function is hosted on a dedicated computer system: the functions are certified once, independently, and only to a level appropriate to the criticality of the function they perform. From this observation it is clear that anything less than full composability will significantly increase the cost of certifying a given avionics suite in an IMA system. Because certification is often a major portion of the cost of an avionics system, any savings in hardware costs will rapidly evaporate in the absence of composability.

The simple fact that the IMA computer system is a shared resource, however, is in direct conflict with the concept of isolation so carefully maintained in the federated architecture. Isolation, such as one finds in a federated architecture, is the key to composability. In order to counteract the lack of physical separation and to achieve the degree of functional isolation required to support composability, an IMA computer system must employ partitioning.

1.3 Partitioning

Partitioning is a technique for providing isolation, both in space and time, between two or more functions executing on the same computer system. A partitioned system provides isolation in space through memory protection and in time through periodic partition switching. A partition is simply a rigid confinement vessel for threads of control and data. This vessel is managed by a partition management system and serves to isolate the behavior each function from the behavior of other functions on the system. Note that this definition of a partition doesn't preclude the use of operating system style processes to implement partitioning. Partitioning is merely a technique for isolating functions in an effort to extend the current federated architecture certification process to include IMA.

1.4 Invariant Performance

Invariant performance is a concept developed to describe the properties of an ideal partitioned system [8]. The invariant performance property is that, given a partition schedule, all aspects of the operation of a given partition are strict functions of the logical state of the partition and its inputs. Invariant performance guarantees that, given identical inputs, the outputs and the execution of a partition are identical regardless of the activity of any other partition in the system.

Invariant performance provides the developer with a contract that states that any function developed in a partitioned environment will operate identically before and after system integration. It also provides the certification authority with the assurance that any test run on a function in a partitioned system will have exactly the same results following system integration. In this

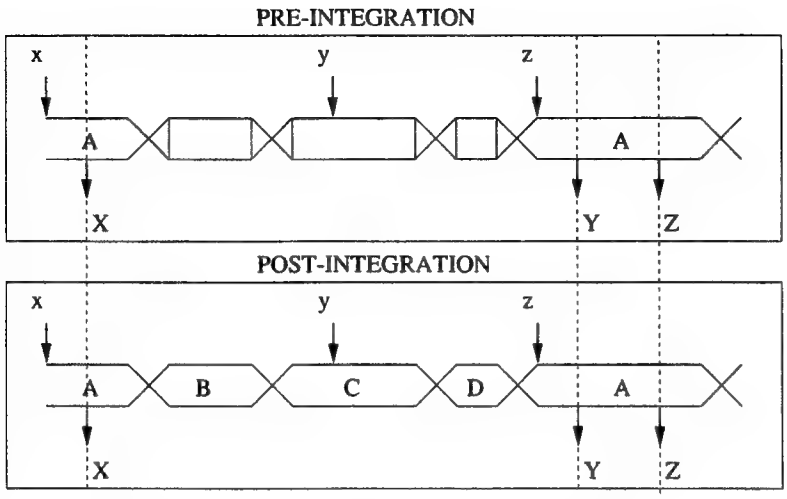


Figure 1: Invariant Performance Illustration

sense, invariant performance provides the strongest possible argument for the composability of integrated modular avionics systems.

Figure 1 illustrates some of the concepts of invariant performance. The upper time line in this figure shows that partition A is scheduled to run during a particular time slot and that it receives inputs (x, y, and z) and produces output values (X, Y, and Z) with a particular temporal relationship. The bottom time shows that, following system integration, if partition A is given the same input values at the same relative time, it produces the same output values and they are made available at the same relative time.

Throughout the remainder of this paper we will consider the ideal partitioning system as one supporting invariant performance. It worth noting that the primary strength of any system supporting invariant performance is that it provides partitioning guarantees in a fashion that is absolutely independent of the actual functions within the partitions. However, if the invariant performance assumptions are weakened, one is drawn into painstaking analysis that may require knowledge and guarantees about the behaviors of functions within partitions. This makes the verification effort more difficult and weakens the composability argument.

1.5 Composability

As previously mentioned, a primary goal of IMA is functional composability: the ability to retain the certification of two functions certified independently when they coexist in a partitioned environment. When considering this IMA objective, it is helpful to understand the distinction between verification and certification.

The verification of a system provides a degree of assurance that the system is built the right way. Verification testing typically includes forcing decision paths through a design in order to show that each path performs as expected and can be reached. The level of verification required by a system is intimately tied to the criticality of the function performed by the system under consideration. The higher the degree of criticality, the more extensive the verification testing must be.

The term "verified" is used throughout this paper to describe a level of confidence that some property holds. Ideally, verified would mean absolutely guaranteed to hold. In practice, however, verified is more likely to mean that the properties in question have been shown to hold in a manner consistent with the criticality of the most critical function to be performed by the portions of the system that depend upon the property. Note that if the correct operation of the partition management system depends upon a property then the entire system depends upon that property.

Certification, on the other hand, is the process by which one obtains credit from a regulatory agency for performing the verification of a system based upon a specific set of verification criteria. Certification involves demonstrating to some degree of confidence that a particular system implements its required functionality. Note that certification only requires assurance proportional to the consequences of failure. In a federated architecture, such consequences are generally limited to the function concerned, so that assurance is related to the criticality of that function. But if the failure of one function could propagate to others, then all must be assured to the level of the most critical. This elevation in assurance levels is contrary to the goal of composability, so partitioning is required when resources are shared by functions that have different levels of criticality and assurance.

Composability is partitioning's "golden ring" and is crucial to the success of IMA. Composability enables incremental certification: each of the software building blocks of a system can be certified independently and only to the level of criticality for the function they perform and then the components can be incrementally combined together without invalidating previous certification work.

It should be noted that, although there is on-going work within SC-182/WG-48 to establish minimum operational performance standards in support of partitioning and computational resources, there have not been enough FAA and JAA certifications of partitioning systems for any sort of standard procedures and expectations to have developed. Nonetheless, composability is an essential enabling technology for IMA. Avionics suppliers will find IMA much less cost-effective if they cannot certify partitions separately. For this reason, the certification process must be of foremost consideration when evaluating a particular IMA partitioning strategy.

1.6 Terminology

Below we provide definitions for selected terminology used in this paper.

Certification (Per SC-167) Legal recognition by the certification authority that a product, service, organization or person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization or person and the formal recognition of the compliance with the applicable requirements by issue of a certificate, license, approval or other documents as required by national laws and procedures. In particular, certification of a product involves: (a) the process of assessing the design of a product to ensure that it complies with a set of standards applicable to that type of product so as to demonstrate an acceptable level of safety; (b) the process of assessing an individual product to ensure that it conforms with the certified type design; (c) the issuance of a certificate required by national laws to declare that compliance or conformity has been found wih the standards in accordance with items (a) or (b) above.

Composability The ability to retain the certification of two or more functions certified independently when they are brought together in an integrated modular environment.

Federated Architecture An avionics system architecture in which each function is provided a dedicated computer system.

Function A generally self-contained program or software system that performs a specific task. Example avionics functions include autopilot, flight management, and displays.

IMA Integrated Modular Avionics. An avionics system architecture that supports the independent certification and composition of multiple functions onto a single computer system.

Invariant Performance A property of a partitioned system that guarantees that, given a partition schedule, all aspects of the operation of a given partition are strict functions of the logical state of the partition and its inputs.

Operating System Those services generally provided by the system to an application, often including resource management, communication and task scheduling.

Partition A space-time container managed by the partition management system that isolates the behavior of a particular function from the behavior of other functions executing on the same computer system.

Partition Management System The combination of hardware and software relied upon to provide and enforce partitioning guarantees.

Partitioning Kernel The privileged, trusted software portion of the partition management system implementation typically responsible for mediating shared resources, supporting communication channels, and partition scheduling.

Verification (Per SC-167) The evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process. Specifically, this includes testing that exercises the conditional paths through a design to an extent defined by guidelines whose rigor is in proportion to the criticality of the function being performed.

2 Overview

In the remainder of this paper we consider the relationship between invariant performance and certification. We will present selected issues faced in the design of a partitioned system and describe how those issues are addressed in an ideal partitioned system having the invariant performance property. As each issue is introduced, relaxations of the invariant performance ideal are considered from the perspective of certification. The relaxations considered are consistent with those one might face in attempting to use commercial components and software to implement a partitioned system. These issues demonstrate how partitioning systems that support invariant performance provide the strongest possible certification argument and how, in the absence of invariant performance, the developer is drawn into partition-specific analysis in order to support incremental certification claims. Note that we are in no way implying that invariant performance is essential for the certification of integrated modular avionics systems. Rather, our claim is that invariant performance is a property that will minimizes the level of effort required by developers to certify such systems.

3 Hardware Components

Ideal The system is composed of hardware components whose behavior is completely specified and verified to be capable of supporting all partitioning requirements.

The component verification process for a partitioned system begins with the process used in federated systems. In addition it must consider those component aspects that impact the system's ability to isolate the various partitions from one another. For processors, issues such as instruction times, interrupt latency, and transitions in and out of privileged modes must be considered. For peripherals, issues such as hidden state and unexpected mode transitions must be addressed. Ultimately, one must verify that the components used in a partitioned system cannot be employed as agents to violate the partitioning requirements.

3.1 Unspecified Behavior and Errata

Verification is straightforward when the components are free of potential violations of the partitioning requirements. However, there may be conditions under which the behavior of a component is unspecified or under which the behavior of

the component is known to violate a partitioning requirement. Note that these two conditions are similar in that unspecified behavior must be assumed to violate partitioning unless other assurances are provided that it will not. When such conditions exist, the issue becomes how to protect against their manifestation. Ultimately, the mechanisms used to protect against component based partition violations may take one of three forms: hardware solutions, single point software solutions, or multi-point solutions.

Hardware Solutions Hardware solutions are solutions that can be implemented in hardware and verified once to work for all systems. Such solutions place no subsequent restrictions on the system software. An example of a component issue that is amenable to hardware solution is Pentium II bug A42 [1], in which the processor may cause bus contention if the bus controller chipset attempts to optimize for arbitration latency. Bus contention, under certain conditions, may qualify as a violation of spatial partitioning. The hardware solution is simply to not optimize for arbitration latency, thus effectively avoiding the issue.

Single Point Solutions Single point solutions are similar in nature to hardware solutions in that they can be fixed once and for all at a single point. However, in this case, the single point is a piece of code included in, perhaps, the partitioning kernel. As an example, the Pentium III errata identifies a BIOS fix for Errata E34 [2], in which certain floating point flags are not updated correctly when values are loaded into cache. While Errata E34 does not necessarily represent a partition violation, it is nonetheless representative of the class of errors with single point solutions.

Multi-Point Solutions Perhaps the most difficult issues to address, and certainly the most common, are those requiring multi-point solutions. Multi-point solutions are solutions that must be implemented in software in more than one location and, often times, directly within the application code stream. Error G39 of the Pentium III Xenon [3] states that a misaligned locked access to APIC space will hang the processor. Hanging of the processor by one partition would certainly be a violation of temporal partitioning. Assuming that locked access to APIC space were for some reason desirable inside of arbitrary partitions, some mechanism must be put in place on a Xenon system to enforce correct alignment. This, however, is an issue in partitioned systems.

In a federated architecture, when a processor contains a bug in a particular instruction, it is standard practice to analyze the software in so as to guarantee that the bug is never exercised or to place software wrappers around the buggy section to detect and correct buggy conditions. This practice works in a federated system because the system is verified as a whole, and the verification of the analysis or the wrapper code is done with consideration of the criticality of the function to be performed.

Partitioning, on the other hand, is designed to support the execution of functions verified to different levels of criticality. Unspecified behavior and errata that potentially affect partitioning guarantees have a global impact on the system. In other words, all of the software on the system depends upon the avoidance of such conditions in every partition. To what extent, therefore, will a level A function accept the analysis performed on the bug fix for a level D function? It seems that the only solution is to perform verification of any such fix to the level of the partitioning kernel, making sure that any assumptions or dependencies encountered during such analysis are also verified to that level.

This elevation in assurance levels, however, is contrary to the goal of composability, which strives to verify each partition only to the level of criticality of the function that it performs. In this way, components suffering from errata or unspecified behavior that require multi-point solutions violate a basic tenant of IMA.

4 Memory Protection Issues

Ideal No instruction executing in user mode can obtain privileged access to the memory space.

Ideal No event can transition control from user mode to privileged mode without transitioning the thread of control to the partitioning kernel.

Ideal The memory protection logic can be modified only in privileged mode.

Most modern microprocessors provide two modes of operation: privileged and user mode. Privileged mode is the least protected mode and offers functionality not available to programs executing in user mode. The nature of the privileged mode functionality is typically such that it impacts the entire system, rather than just the current thread of control. This functionality is therefore restricted to privileged threads of control, such as the operating system, in order to protect arbitrary user tasks from one another. Memory management schemes are also typically sensitive to the processor mode and will allow memory transactions in privileged mode that would be illegal in user mode, including access to the memory space of other programs. It is easy to see why, in a partitioned system, one would require that access to this mode of operation be controlled.

A significant issue in the design of a partitioned system is allowing both the partitioning kernel and the operating system to execute in privileged mode. If a processor supports two or more privileged modes of operation, it may be possible to implement the partitioning kernel in a more privileged mode than the operating system. When this is not possible, however, no distinction is made between the access rights of the OS and those of the partitioning kernel. As a result, the partition management system is made vulnerable to errors in the operating system. Because of the dependencies introduced by this fundamental limitation, the entire OS must be verified together with and to the same level as the partitioning kernel.

This issue is particularly important when one considers a virtual machine style partitioning scheme that allows different operating systems to execute in different partitions, with each executing in privileged mode. If this is the case, all of the operating systems must be verified to the highest level and they must all be verified together with and to the same level as the partitioning kernel. Note that this in direct violation of the IMA composability criteria.

5 Temporal Protection Issues

Ideal Partition Timers are protected resources and only the partitioning kernel can modify the timers.

Ideal The partition timer interrupts are non maskable interrupts that can be modified only by the partitioning kernel.

Allowing partitioned software to modify the partition timer, the partition timer interrupt or the partition timer interrupt mask provides the opportunity to violate temporal isolation. Any software with such capability must be certified together with and to the same level as the partitioning kernel.

Ideal The processor must respond to the partition timer interrupts within some acceptable, quantifiable time period.

Unbounded partition timer interrupt latency is a violation of temporal isolation. If the processor contains instructions with unacceptable or unquantifiable interrupt latency or if the processor can enter a state in which it does not respond correctly to the partition timer interrupt, some mechanism must be provided to force the processor back into a known state from which partition scheduling can be resumed. It may be the case that the only available mechanism is a hardware reset. In any case, the mechanism must then be integrated into a partition watchdog timer that is capable of monitoring the health of the partition management system. Note that all of the aforementioned restrictions applying to the partition timer apply to this watchdog timer as well. Furthermore, the time required to resume correct partitioned execution following watchdog time-out must be quantifiable and included in the partition context switch overhead.

5.1 Performance Issues

Ideal For a given partition schedule and a given set of inputs, the execution of a given partition is exactly the same, to the clock cycle, regardless of the activity of any other partition.

Performance guarantees relate to reproducibility of results in partitioned systems. The ideal system is absolutely reproducible (modulo any asynchronous inputs) by virtue of clock cycle accuracy. This guarantee builds confidence that the system will behave as expected, even following integration, and provides

both the developer and the certification authorities with a contract that guarantees that the system will operate exactly the same, both on the test bench and in the system.

5.1.1 Instruction Times

Invariant performance expects that instruction execution time in the context of the system be a function of only the logical state of the partition in which it executes. In other words, a partition should execute the same number of instructions during any fixed period of time for a given initial logical state. It is, however, a simple matter to violate this assumption. The mere existence of a cache, while it does not change the logical state of a partition, can change the processor performance within a partition from one partition execution to the next. If at one time the cache is full and in the next it has been flushed, the partition will execute a different number of instructions during some fixed length of time. A cache is an example of hidden system state that can allow execution in one partition to impact the performance of another partition. Hidden state and dynamic bus arbitration are common causes of system nondeterminism. If one does not know or cannot control the parameters affecting the execution time of the various instructions, it is impossible to guarantee that the execution of a function within a partition will be exactly the same following a partition switch.

5.2 Uniform Start Times

Ideal All partitions begin execution at exact times relative to the partition interrupt.

Ideal Early exit from a partition will not affect the start time of a subsequent partition.

In order to be compliant with the statement of invariant performance, the partitioning kernel must perform partition execution dispatch at uniform times. Note that this is an unusual requirement and one that will almost always require hardware support. Typical real-time systems provide the option of uniform interrupt intervals. However, variable interrupt latency, coupled with non-trivial (thus difficult to analyze) context switching, will typically lead to non-uniform partition start times, even under the best of circumstances. Uniform start times typically require a second synchronizing event following the partition switch interrupt that can be used to absorb the random fluctuations normally associated with partition context switching. This second event can then be used to provide a uniform starting point for partition execution.

Figure 2 illustrates the concept of uniform start times and the various conditions under which it must hold. In the upper time line partition A exhibits a variable interrupt latency which must be completely absorbed by the partitioning system. In the lower time line partition A exits early due, perhaps, to an error in that partition. Note, however, that the kernel does not dispatch partition B until partition B's start time arrives.

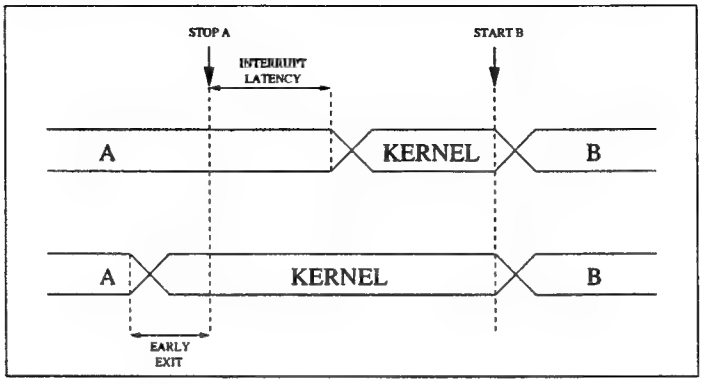


Figure 2: Uniform Start Time Illustration

Failure to provide uniform start times under any condition violates invariant performance and necessitates the quantification and bounding of partition schedule jitter. These bounds must then be addressed as a part of the process of evaluating and verifying the real-time behavior of each function.

5.2.1 Performance Bounds

For many systems it is arguably the case that exact temporal behavior is not necessary for functional correctness. Most real-time operating systems today provide some form of multitasking, but few if any make claims beyond statistical guarantees of service, latency or jitter. Even given deterministic operating system task scheduling, applications still face all of the issues associated with variations in instruction execution time; a problem exacerbated by caches, pipelines, and speculative execution, all of which substantially improve average performance at the expense of worst case performance.

With this as the current state of affairs, it could be argued that the concept of invariant performance is asking for properties of a partitioned system that are above and beyond what current practice provides in a federated system. Furthermore, the variations in performance due to flushed caches or jitter in the partitioning schedule could simply be modeled in the same way as the "natural" variations in a federated system described above. This would then lead to a straight-forward extension of current practice.

While this is certainly a valid argument, it should be observed that invariant performance, far from being an outlandish requirement, is in fact the typical state of affairs in most federated systems today. Although it is true that fluctuations exits in the execution time of various instructions and that jitter is an accepted aspect of the scheduling mechanisms, these variations are all reproducible in a federated system modulo asynchronous events. In a completely

synchronous system, these events are reproducible exactly. Thus, in effect, federated systems already provide invariant performance.

In a partitioned system, however, when these so-called natural events are played upon by different functions executing in different partitions, the result is a temporal variation that cannot be accounted for nor reproduced by considering only the execution of the partition of interest. Keep in mind that the goal of partitioning is composability: the ability to certify functions independently and then combine them together to create a certified system. The key to this ability relies on partitioning to isolate the behavior of different partitions in such a way as to emulate the physical isolation of a federated system.

Nonetheless, in the final analysis strict temporal partitioning may be deemed unnecessary. While it is certainly the case that invariant performance is beneficial in a synchronous environment, most useful systems are ultimately asynchronous. Given this, the designer is still left with the task of producing assurances based on the expected operating environment that can provide a guarantee of correct functional operation.

6 Partition Scheduling

The partitioning kernel is responsible for allocating and protecting shared resources among the various partitions. One shared resource in a partitioned system is processor time. Time is allocated to partitions by alternately suspending the dispatching partition execution.

6.1 Schedule Analysis

Ideal The scheduling policy is amenable to deterministic analysis and that the analysis is independent of the behavior of the partitions to be scheduled.

Ideal The latency and throughput of each function can be computed knowing only the partition schedule.

Figure 3 illustrates the impact of partitioning on the latency and throughput of a typical function. The function executes within the time constraints of a partition, labeled A in the diagram. The time it takes for the function to begin to respond to an input is the response latency. The time it takes to service the input and produce some output is the service time. Both of these aspects will be negatively effected by partitioning. In the ideal system, these effects are known and quantifiable at the individual partition level. If exact analysis is not possible or if the scheduling is somehow function-dependent, it must be possible to provide absolute bounds on the maximum latency and minimum throughput available to each function. These bounds must then be addressed as a part of the process of evaluating and verifying the performance and spare capacity of each function.

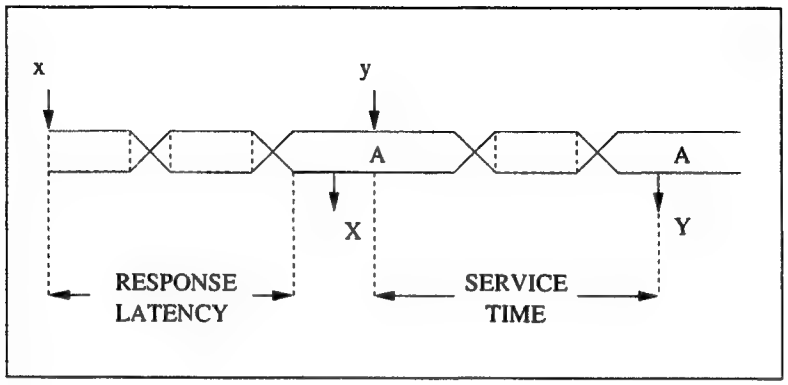


Figure 3: Performance and Latency Impact of Partitioning

6.2 Reset

Ideal On reset, the partitioning kernel performs, at most, partition independent initialization of each function and then signals each partition to initiate partition dependent initialization.

Reset, or cold start, corresponds to a complete reboot of the system. Starting with only the information available in non-volatile memory, the processor must first bootstrap the system to the point at which software can begin executing. That software must then complete the configuration of the system and ultimately begin running the various functions. This sequence of events is referred to as the reset sequence.

Obviously, once a partition has been configured for execution by the partitioning kernel, each function must be capable of bootstrapping into a state in which it can execute successfully. If multiple functions exist on the same computing resource, it must be possible to bootstrap all of the functions.

In the simplest case, the partitioning kernel is the only software responsible for initializing a partition. However, more sophisticated systems may have boot loader hierarchies that gradually increase in complexity and become increasingly more application dependent. Note, however, that any software responsible for initializing a partition must be verified to the level of the partition being initialized.

Of greater concern, however, is whether partition initialization is being performed in privileged mode. If this is the case, one must verify that initialization itself does not violate partitioning. Any function-specific initialization that is performed in privileged mode must be verified together with and to the same level as the partitioning kernel. Unless care is taken, this may lead to the

undesirable scenario in which the partitioning kernel will have to change (and therefore be re-verified) to support different system configurations.

6.3 Power Down and Warm Start

Ideal On power down and warm start, the partitioning kernel may modify the partition schedule to accommodate power down timing requirements and then notify each concerned partition in some modular fashion of the pending event.

Power Down and warm start are examples of global system events for which each function may require its own handling procedure. The requirements for power down may vary significantly from one function to the next and it is not desirable to have partition specific kernel code handling such diverse requirements. Partition specific code in the kernel may force recertification of the kernel for different system configurations, ultimately hampering composability.

7 Operating System Versus Partitioning Kernel

In a partitioned system, the distinction between the operating system and the partitioning kernel is not always clear. Figure 4 illustrates two common models for partitioned systems: the virtual machine model and the tasking model [7]. In the virtual machine model, both the applications and the operating system are isolated from the partitioning kernel. Each partition, therefore, has its own operating system and acts as a virtual machine. In the tasking model, the partitioning kernel and the operating system are essentially one and the same. In this model, partitioning is performed at the task level. While the virtual machine model allows for a smaller, simpler partitioning kernel as well as a higher degree of isolation between partitions, the tasking model may be more practical in commercial systems due to restrictions of commercial operating systems or to the limited number of privileged modes supported by commercial processors.

Regardless of the degree to which a particular implementation matches one of the two models, however, isolation conflicts will inevitably exist between the operating system and the partitioning kernel. Three possible isolation conflicts are in the areas of data handling and data structure access, exception and error handling, and interrupt handling.

7.1 Data Structures and Data Handling

Ideal The partitioning kernel data structures can be modified only by the partitioning kernel. The partitioning kernel never relies on data that does not originate exclusively from the partitioning kernel memory space.

When the operating system operates in privileged mode, it effectively has access to all of the partitioning data structures. Any software, including the

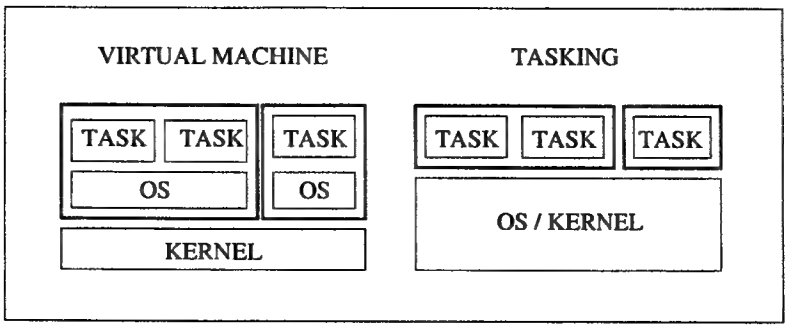


Figure 4: Partitioning Models

operating system and partitioning kernel, that executes in privileged mode must take care when using data from within an arbitrary partition's memory space. If privileged mode software ever accepts pointers, for example, from within a partition's memory space, the privileged software must be verified to the level of the partitioning kernel to operate correctly, even in the presence of corrupted or malicious pointer values.

If the correct operation of the partitioning kernel relies on the correctness of data from a memory space that can be modified by partitioned code, any software executing in that partition must be verified together with and to the same level as the partitioning kernel.

7.2 Exceptions and Errors

Ideal All exceptions and errors are handled outside the partitioning kernel and so as not to impact the operation of the kernel. The partitioning kernel is verified to be free of any exceptions or errors.

Exceptions and errors are events in the processor that cause an otherwise unexpected transfer of control. While the response of different processors to such events varies significantly, it is important in a partitioned system to be able to isolate the results of exceptional and error events to the partition in which they occurred.

This isolation may be particularly challenging for exceptions or errors that propagate into the privileged operational mode of the processor. Because privileged mode operation may be uninterruptible, these error handling sections must be carefully considered when computing temporal properties of the system. Many of the issues associated with exceptions and errors also arise when considering interrupts, which are discussed in Section 7.3.

A system must be careful not to violate partition isolation when considering attributing exceptions and errors that propagate into (or originate from) the partitioning kernel to particular partitions. An error must be attributed to a particular partition only if the error would have occurred regardless of the contents of the other partitions.

7.3 Interrupt service models

Ideal At any given instant, each interrupt in a partitioned system is owned by a particular partition or by the partitioning kernel. Interrupts are acknowledged only while the owning entity is executing.

Invariant performance mandates that the activities associated with a particular partition do not impact the operation of any other partition. The activities associated with a partition include those activities involved in servicing interrupts associated with the partition. While the ideal may be unattainable in some systems, it is nonetheless crucial to understand the impact interrupt servicing has on the partitioned system. Towards this goal, we broadly classify the possible interrupt servicing schemes according to two criteria : the time at which the interrupt service is provided and the degree to which the partitioning kernel is involved.

7.3.1 Service Timing

The timing of the interrupt service has a direct impact on the temporal properties of the partitioned system. It is often the case that interrupt handlers are, themselves, uninterruptible. This fact can lead to increased interrupt latency which can effect both function performance and partitioning guarantees. It is also important to be able to quantify the impact interrupt servicing has on the performance and latency of individual functions so that developers can compute such critical factors as spare throughput and response time. There are three basic approaches for timing interrupt services in a partitioned system: immediate, interstitial, and targeted.

Immediate Immediate interrupt handlers service interrupts as soon as possible, regardless of the partition schedule. While this model of interrupt service is quite natural for federated systems, in a partitioned system this interrupt policy results in system wide performance variations due to interrupt activity that may functionally impact only a subset of the partitions. The result of this global impact is strict limitations on the number and frequency of interrupts allowed by the system, a limitation which might ultimately have to be enforced in hardware.

Immediate interrupt servicing has a negative impact on the throughput and latency available to every function in the system, an impact which must be considered when computing the processing requirements of individual functions. At a minimum this impact must be quantified and bounded and these bounds

must then be addressed as a part of the process of evaluating and verifying the real-time behavior of each function.

Interstitial Interstitial handlers service interrupts only while performing partition context switches, in the interstices between partitions. Such a design eliminates the impact of interrupts on the performance of each function but can result in a significant increase in the partition switch latency. Nonetheless, this style of interrupt handling can be readily accommodated using interrupt controllers with simple interrupt masking capabilities.

Targeted Targeted interrupts are interrupts that have been "targeted" or linked to a particular partition. The handlers servicing targeted interrupts are associated with a particular partition and execute only when the partition owning those interrupts is scheduled. Targeted interrupts are advantageous in that their temporal impact is localized, since a targeted interrupt owned by one partition has no impact on the timing behavior of any other partition. Thus, the frequency and duration of such interrupt handlers will affect only the partition owning those interrupts. Targeted interrupts, however, require that the interrupt mask logic be managed by the partitioning kernel during context switches so as to avoid allowing one partition to steal an interrupt from another and to protect partitions from interrupts generated in other partitions. The hardware used to support targeted interrupts must insure that residual interrupt state does not carry over from one partition to another. Hardware induced latency and hidden interrupt state may make the use of standard commercial interrupt controllers impractical for such applications.

7.3.2 Partitioning Kernel Involvement

The degree of partitioning kernel involvement in servicing interrupts has a significant impact on the extent to which interrupt handlers and the partitioning kernel can be verified independently. In many systems, the degree of partitioning kernel involvement is dictated by the interrupt behavior of the processor. Many commercial processors automatically enter privileged mode when servicing interrupts. Although a transition into privileged mode need not imply execution of the partitioning kernel thread, it does compromise the isolation of the partitioning kernel from the interrupt handlers.

Full Service Full service kernel handlers completely handle the interrupt and then communicate the result to the associated partition. This makes the partition code very portable but places a heavy burden on the partitioning kernel, a burden which may result in increased partition interrupt latency and a heavy drain on available throughput. Such a service model tends to result in an function-specific partitioning kernel with limited composability.

Lightweight With lightweight handlers, the kernel interrupt handler merely communicates the event to the associated partition, preferably in some modular fashion. The majority of the interrupt processing occurs when the relevant partition is subsequently scheduled. This approach minimizes the function-specific code in the partitioning kernel but increases the system dependency of the partition code. This approach is composable to the extent that it is possible to signal partitions of pending events in a modular fashion.

Transparent A transparent interrupt handler does not involve the execution of any partitioning kernel code and the processor never enters privileged mode. This solution may not be an option with most commercial microprocessors since many processors enter privileged mode while servicing interrupts. However, such an approach does provide maximally independent partitioning kernel code at the expense of heavy system dependence in the partition operating system code.

7.4 Other Interrupt Issues

The scope of different interrupts may present particular issues in the design of interrupt controllers and interrupt handlers. Interrupts can be classified into three categories according to their scope: partitioned, shared, or global.

7.4.1 Partitioned Interrupts

A partitioned interrupt is an interrupt corresponding to a resource that is owned exclusively by one partition. An example of such an interrupt might be one originating from a dedicated UART. Because the partitioned interrupt is intended for only a particular partition, invariant performance dictates that interrupts effect only the owning partition and that no other partition is capable of stealing the interrupt events from the owning partition. As previously discussed, however, the extent of the impact of any such interrupt depends heavily upon the interrupt service style adopted by the system.

7.4.2 Shared Interrupts

A shared interrupt is an interrupt corresponding to a single resource that is time shared among the various partitions. An example of such an interrupt is a transaction time-out interrupt originating from a memory controller. If a transaction is initiated that causes the bus to hang, the bus arbitration logic may terminate the transaction and generate a transaction time-out interrupt to the processor. Note that such an event is the result of the execution of a particular partition (assuming it was not a result of kernel execution) and that the event must be communicated exclusively to the offending partition. Shared interrupts are of particular concern around the time of a partition switch. Shared interrupts, if not managed properly, can lead to event leakage between partitions and ultimately compromise system composability.

7.4.3 Global Interrupts

Certain resources are shared among all of the partitions in a multi-partitioned system simply by virtue of their existence on the same processing resource. Examples of such resources are clock (time), power supply, thermal condition, EMI, ionizing radiation exposure, etc. Because of the existence of these resources, certain functionality may be needed to support activities surrounding these resources. However, such functionality may differ on a partition by partition basis.

A global interrupt is an interrupt corresponding to a shared resource that affects multiple partitions residing on the system. Examples of such interrupts are reset, partition switch, power down, and warm start. Global interrupts are characterized by necessitating action from the partitioning kernel, from two or more partition resident in the system, or from both the partitioning kernel and one or more other partitions.

It is important from the perspective of certification that the kernel-specific code be independent of the partition specific code. Optimally, the kernel-specific code is executed in kernel mode and the partition-specific code is executed in the context of the partition to which it belongs. This allows independent verification of the kernel and partition code and also serves to isolate the partition specific code from other partitions. The inability to modularize such event handlers leads to a breakdown of composability.

8 Resource Arbitration

Ideal Each system resource required by each function previously hosted on a federated system must be duplicated for each function instance composing an integrated system [4].

Adherence to the above stated ideal eliminates all but the intrinsically shared resources (time, power, etc). However, it also goes against one of the partition motivating factors: reduced component costs. From this perspective, it is preferable to share as many redundant system resources as possible.

If the partitioned system provides a resource that is shared between partitions in an exclusive way, the system must also provide an arbitration service for that resource in order to maintain an acceptable degree of functional isolation between those partitions. The arbitration service must decide when to allocate a resource, for how long to allocate the resource, and must be capable of forcibly removing control of the resources from any non-compliant partitions. This arrangement abstracts the interdependency between functions in different partitions and provides fault isolation between them.

One means of providing arbitration services for resources such as communication ports is to encapsulate the services within the context of an independent partition. This "broker" partition would accept resource requests from other partitions in the system, allocate resources to the partitions based on some scheme, and monitor resource usage to provide fault isolation. Of course the

broker partition must still be certified to a high level of criticality, but this scheme provides the advantage of isolating resource arbitration from both the individual partitions as well as the partitioning kernel.

If arbitration services are not provided at the system level for a particular shared resource, all of the partitions competing for that resource must be verified together to the highest collective degree of criticality.

Regardless of how the arbitration service is provided, the latency and throughput bounds imposed by the arbitration must be quantifiable and must be considered when analyzing the latency of and throughput requirements of the functions relying on that resource. If the arbitration scheme cannot guarantee performance bounds on the resource, all of the partitions sharing that resource must be verified together to the highest collective degree of criticality.

Note that the shared resource requirements of each function must be carefully analyzed to guarantee schedulability given an arbitration scheme, exactly as the processor throughput and latency requirements of each function in a partitioned system must be analyzed to guarantee schedulability on the shared host processor. There is no substitute for this analysis. No hardware or software partitioning support is capable of isolating functions that require use of a shared resource and whose combined performance requirements exceed the capacity of that resource.

While the above discussion is geared towards such shared resources as communication ports or peripheral devices, they also apply to more obscure resources such as multi-master busses and data and instruction caches. While the details are more involved, the bottom line is that the performance impact of such resources must be completely deterministic if they are to support invariant performance. If not completely deterministic, the behavior must be bounded in such a way as to allow credible analysis of behavior, performance, and latency. In the absence of such well defined behavior, all of the partitions sharing that resource must be verified together to the highest collective degree of criticality.

9 Autonomous Hardware Agents

Ideal The central processing unit is the only agent capable of modifying a partition's logical state.

One can consider any automata other than the central processing unit that performs actions on the system to be an autonomous agent. Such hardware agents may act either on behalf of a particular partition or independently of any particular partition. A Direct Memory Access (DMA) unit is an example of an agent that operates on behalf of a specific partition. A DRAM refresh circuit might be an agent acting independently of any particular partition.

Agents can impact system behavior in many different ways. They may modify the logical state of partitions or perhaps arbitrate with partitions for shared resources. Agents that modify memory or steal bus or CPU cycles must be verified to do so under the constraints of the partitioned system. An agent can

change the logical state of a partition only if the partition expects the agent to do so.

Agents that arbitrate for resources must do so in a fashion that is independent of the partitions resident on the system and in a completely deterministic manner if they are to satisfy invariant performance. In the worst case there must be a quantifiable bound on their performance impact and that bound must be addressed during the verification of each application.

10 Conclusion

This paper illustrated the concept of invariant performance in a partitioned system and explained how it provides the strongest possible composability argument from the perspective of certification. The strength of invariant performance is that it allows one to provide partitioning guarantees in a fashion independent of the actual functions within the partitions. We explained how, as the assumption of invariant performance is weakened, one is drawn more deeply into analysis that requires knowledge and guarantees about the behaviors of functions within partitions. This makes the verification effort more difficult and weakens the certification argument. While practical issues may limit a system designer's ability to provide invariant performance partitioning using commercial components, ATC has recently developed a system based on the JEM2 and a companion partition management unit (PMU) that provides invariant performance in a partitioned system.

References

[1] Intel Corporation. Pentium II processor specification update. Technical report, Intel, 1999. Order No: 243337-027, Available at http://developer.intel.com.

[2] Intel Corporation. Pentium III processor specification update. Technical report, Intel, 1999. Order No: 244453-004, Available at http://developer.intel.com.

[3] Intel Corporation. Pentium III Xenon^{TM} processor specification update. Technical report, Intel, 1999. Order No: 244460-003, Available at http://developer.intel.com.

[4] John Gee. Multi-function embedded computers. Technical report, Advanced Technology Center, Rockwell Collins. Work in Progress.

[5] David S. Hardin. A multipartitioning system for avionics applications. Technical report, Advanced Technology and Engineering, Collins Commercial Avionics.

[6] David S. Hardin. Partitioning system requirements and architecture. Technical report, Advanced Technology Center, Rockwell Collins, Inc.

[7] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, October 1998. Unpublished draft report; available at http://www.csl.sri.com/ rushby/partitioning.html.

[8] Matthew Wilding, David Hardin, and David Greve. Invariant performance: A statement of task isolation useful for embedded application integration. *Proceedings of Dependable Computing for Critical Applications - DCCA-7*, 1999.

Bibliography

[1] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, NASA Langley Research Center, Hampton, VA, June 2000. Proceedings available at `http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/`.

[2] E. A. Emerson and A. P. Sistla, editors. *Computer-Aided Verification, CAV '2000*, Volume 1855 of Springer-Verlag *Lecture Notes in Computer Science*, Chicago, IL, July 2000.

[3] David Greve. Symbolic simulation of the JEM1 microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, Volume 1522 of Springer-Verlag *Lecture Notes in Computer Science*, pages 321–333, Palo Alto, CA, November 1998.

[4] David Greve. Invariant performance and commercial system components. Unpublished manuscript, 1999.

[5] David Greve. Invariant performance and PMU design considerations. Unpublished manuscript, 1999.

[6] David Greve and Matthew Wilding. Invariant performance and PVS: A statement of task isolation that can be certified by machine-checking. Unpublished manuscript, 1999.

[7] David Greve and Matthew Wilding. PVS code proofs: Benchmarking and enhancements. Unpublished manuscript, 1999.

[8] David Hardin. Partitioning system requirements and architecture. Unpublished manuscript included in the Interim Report for Contract F30602-96-C-0204, Arpa Order D855, September, 1998, 1998.

[9] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In Emerson and Sistla [2], pages 521–537.

[10] John Rushby. Ubiquitous abstraction: A new approach for mechanized formal verification (extended abstract). In John Staples, Michael G. Hinchey, and Shaoying Liu, editors, *Second International Conference on Formal Engineering Methods (ICFEM '98)*, pages 176–178, IEEE Computer Society, Brisbane, Australia, December 1998.

[11] John Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Available at `http://www.csl.sri.com/~rushby/abstracts/partition`, and `http://techreports.larc.nasa.gov/ltrs/PDF/1999/cr/NASA-99-cr209347.pdf`; also issued by the FAA.

[12] John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In Emerson and Sistla [2], pages 508–520.

[13] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000: Concurrency Theory*, pages 1–16, State College, PA, August 2000.

[14] Natarajan Shankar. Symbolic analysis of transition systems. In Yuri Gurevich, Phillip W. Kutter, Martin Odersky, and Lothar Thiele, editors, *Abstract State Machines: Theory and Applications (ASM 2000)*, pages 287–302, Monte Verità, Switzerland, March 2000.

[15] Natarajan Shankar. Static analysis for safe destructive updates. Unpublished manuscript, 2001.

[16] Mandayam Srivas. Automating partition proofs. Unpublished manuscript, 2000.

[17] Matthew Wilding. Invariant performance. Unpublished manuscript included in the Interim Report for Contract F30602-96-C-0204, Arpa Order D855, September, 1998.

[18] Matthew Wilding, David Greve, and David Hardin. Efficient simulation of formal processor models. *Formal Methods in Systems Design*, 18(3):233–248, May 2001.

*U.S. GOVERNMENT PRINTING OFFICE: 2002-710-038-10256

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.